VMR^{2}L: Virtual Machines Rescheduling Using Reinforcement Learning in Data Centers

Abstract

Current industry-scale data centers receive thousands of virtual machine (VM) requests per minute. Heuristic algorithms are used to allocate these requests in real time to different Physical Machines (PMs). However, due to the continuous creation and release of VMs, heuristic algorithms suffer from inefficient resource utilization, e.g., many small resource fragments are scattered across PMs. To handle these fragments, data centers periodically reschedule some VMs to alternative PMs.

This paper investigates the VM rescheduling problem. We first show that even during off-peak hours, the running time of a VM rescheduling algorithm has critical impact on its performance. Most off-the-shelf combinatorial optimization algorithms scale poorly under such a strict low-latency requirement. Therefore, we develop a reinforcement learning system for VM rescheduling, VMR^{2}L, which incorporates a set of customized techniques, such as a two-stage attention-based framework that handles a variety of constraints and workload conditions, effective feature extraction, and a risk-seeking evaluation module that allows end-users to flexibly balance between latency and accuracy. We conduct extensive experiments with datasets generated from an industry-scale data center. The results show that VMR^{2}L can solve the VM rescheduling problem by comparable performance as the optimal solution, but with a running time of seconds.

1 Introduction

Cloud service providers allow end-users to access computing resources, such as CPU and memory. They adopt resource virtualization to maximize hardware utilization, allocating Virtual Machines (VM) [13, 62] with the requested resources to end-users. An industry-scale data center typically has hundreds to thousands of Physical Machines (PMs), where each PM can host multiple VMs that run independently [25]. A central server manages all VM requests on PMs by performing two tasks, scheduling and rescheduling. When a new VM request arrives, the scheduling algorithm assigns it to a proper PM for different resource utilization goals, such as minimizing the overall fragment rate (FR) or maximizing the number of available PMs. For example, when the remaining resources on a PM cannot fulfill a VM request of X CPU cores, $X \in \{2,4,8,16,32\}$, a fragment occurs. FR is the ratio between the remaining CPU resources that cannot be further utilized by a X-Core VM and the total free CPU resources across all PMs.

An optimal scheduling algorithm needs to consider all the VM requests in an optimization problem. However, as the demand for computing resources increases, hundreds of VMs can enter and exit a data center per minute. The execution time to solve the resource allocation problem is too long to process all the requests in real time. Simple heuristic algorithms [12, 16, 24, 46] are normally used, but they are not optimal and may result in many fragments scattered across PMs. Therefore, the VM manager needs to periodically migrate some VMs to alternative PMs to reduce FR. As a result, rescheduling is critical to optimize resource usage. Taking a datacenter of two PMs as an example, each PM has 44 CPUs. PM1 has 24 allocated CPUs and 20 free CPUs. PM2 has 32 allocated CPUs and 12 free CPUs. The FR is calculated as ((20%16)+(12%16))/32=50%. X is set to 16, as VMs requesting 16 or less cores are the most popular specifications sought by end-users. In our dataset from a commercial data center, these VMs account for 88.87% of all VM requests. The cloud service providers want to save the available PM resources for future 16-core VMs. Although the total number of free CPUs on two PMs is 32, they can only serve one more 16-core VM on PM2, resulting a FR of 50%. Assuming there is a 4-core VM on PM2, we can reschedule this VM from PM2 to PM1, reducing FR to (16%16+16%16)/32 = 0%. All free CPUs can now be utilized for up to two 16-Core VM requests. By simply rescheduling one VM, we can double the number of new 16-core VM requests that can be served.

We can formulate VM rescheduling as a Mixed Integer Programming (MIP) problem that finds the best VMs and migrates them to proper PMs for optimizing resource utilization. The problem may have some constraints from the service

expectations (e.g., the maximum number of migrated VMs) and the available hardware resources (e.g., current number of allocated VMs and occupied PMs). An off-the-shelf MIP solver, such as Gurobi [2] and CPLEX [1], can be used to solve the optimization problem. However, solving a large-scale MIP problem costs minutes or even hours, especially for large data centers beyond thousands of VMs and PMs. Such a long running time is intolerable, since new VM requests will have arrived during this delay, causing the generated solution to be no longer optimal or even feasible.

To accelerate the running speed of the MIP approach, hand-tuned heuristics can be integrated into the process, e.g., adding constraints to limit the solver's search space. These heuristics must trade-off between the optimality of the solution and the tractability of the problem. Unfortunately, even highly-skilled experts need many iterations to find a proper trade-off manually. Moreover, no universal heuristics that can achieve a good trade-off for all VM rescheduling scenarios. Operators have to manually examine and repeat the iterative design process for every scenario.

In this work, we develop VMR^{2}L, a deep Reinforcement Learning (RL) system for VM rescheduling. VMR^{2}L trains a Deep Neural Network (DNN) as the rescheduling agent that learns an optimal rescheduling algorithm by interacting with a data center. The DNN agent takes the current state of the data center as input, and outputs a pair of VM and PM as action (i.e., migrating the VM to the PM). Such a DNN inference can be done within 10 ms. We can run the inference multiple times to move a certain number of VMs to new PMs. To avoid the cost of training a VM rescheduling agent in a real data center, we use a simulator that generates the next state (the VM mapping over all PMs) according to the current state and the VM migration action. As a result, VMR^{2}L can provide comparable rescheduling performance as the MIP solver, but only within 1 second. To transform VMR^{2}L into a practical system, we need to tackle the following three challenges.

First, a commercial data center needs to handle thousands of VMs over hundreds of PMs. An action of VMR^{2}L is to find a VM and migrate it to a proper PM. The action space is large, e.g., almost millions of combinations. It's hard for RL agent to explore such a large action space and learn a good policy [38,40]. To handle this problem, we design a two-stage VM rescheduling pipeline. The first stage selects a VM to be migrated and masks out all the PMs that cannot host the selected VM due to some constraints, such as insufficient CPU resources or affinity between VMs. The second stage chooses the appropriate destination PM to host the selected VM. Additionally, we also inject domain knowledge from conventional methods by pretraining the first stage via imitation learning.

Second, VMs continually enter and exit data centers, leading to a dynamic number of VMs deployed on PMs; however, VMR^{2}L's rescheduling agent takes the mapping of all VMs over PMs as the input. Due to a dynamic number of VMs, the state's dimension to the RL agent changes too. To handle this problem, we leverage a transformer-based neural network to extract effective features from raw data. Notably, we recognize the VM rescheduling problem as reassigning edges on a bipartite graph composed of two-level trees. To incorporate the graph information into our model, we introduce a sparse attention within each local tree and redesign the transformer network accordingly.

Third, The randomness in the policy leads to sub-optimal trajectory sampling and affects the final performance. To guarantee the optimal trajectory selection, we note that our VM rescheduling problem is a special case in reinforcement learning. Given a state and an action, the next state is deterministic, i.e., from an initial state and a sequence of actions proposed by the RL agent, we can simulate the exact reward that would be obtained if these actions were executed. In light of this, we analyze how sampling multiple action trajectories and only deploying the one that leads to the highest reward affects policy gradient-based algorithms from a risk-seeking perspective.

Finally, we implement a prototype of VMR^{2}L including the above three key components. Extensive evaluation on two real datasets demonstrates that The results show that under a typical workload, VMR^{2}L can generate a solution within one second and its solution is only 2.67% worse than the optimal solution. We also verify that VMR^{2}L can generalize to different workloads and additional objectives beyond FR.

We summarize the contributions of this paper as follows:

- **RL for VM rescheduling.** We formulate the VM rescheduling problem as a Markov Decision Process (MDP), and develop a RL-based system, VMR^{2}L, to solve this problem.
- **Customized RL techniques for VM rescheduling.** We tackle three challenges in designing a RL framework for VM rescheduling and tackle them by three customized RL techniques, including two-stage VM rescheduling pipeline, effective feature extraction, and risk-seeking evaluation of the VM rescheduling agent.
- **A VMR^{2}L prototype and extensive evaluation.** We develop a prototype of VMR^{2}L and conduct entensive experiments over two real datasets. We release the datasets and a custom Gym environment for RL training. Our datasets and code are available here[1].

2 Background

2.1 VM Scheduling and Rescheduling

Cloud service providers offer VMs to users for their computation requests. This process involves ① VM Scheduling and ② Rescheduling stages shown in Figure 1.

[1] https://github.com/RLVMR/VMR2L-NSDI24

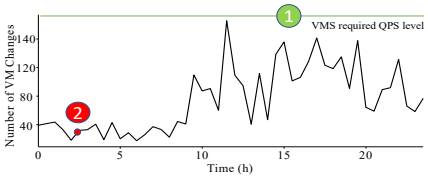


Figure 1: VM request generation speed.

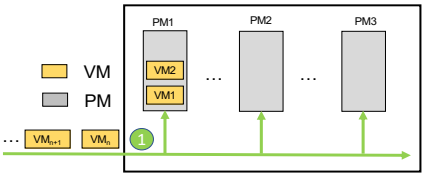


Figure 2: VM scheduling procedure.

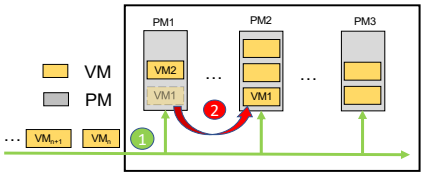


Figure 3: VM rescheduling procedure.

VM Scheduling (VMS). When new VM requests arrive, there are multiple available PMs that can host them. A typical solution exploits heuristics based on bin packing. VMs and PMs are objects and bins respectively. For instance, the first-fit algorithm [12, 16] traverses all the PMs and places the VM on the first PM that can meet the CPU and memory requirement of the VM request. The best-fit algorithm [24, 46] sorts all PMs that can meet the requirements of the current VM according to the amount of the FR reduction before and after this VM is added, and schedule the PM with the largest FR reduction. The best-fit method is currently used in **anonymous company**'s data centers to reduce FR.

VM Rescheduling (VMR). The VM rescheduling algorithms migrate VMs from their source PMs to destination PMs. Due to the overhead of VM migrations, a number limit is set to control the number of VMs to migrate. At **anonymous company**, whenever a high FR is observed that could potentially lead to insufficient resources for upcoming VM requests, a round of VM rescheduling is initiated.

The Necessity of Rescheduling. Commercial data centers need to handle thousands of VMs (queries) arriving and exiting per second (QPS) with a high processing throughput. We collect a 30-day dataset from an **anonymous company** data center including the number of VMs changes (VMs arriving and exiting) per minute. We also do experiments with the data from a large-scale data center in Section 4. Figure 1 depicts the maximum number of VM changes traced per minute over 24 hours. To ensure scheduling is robust, the VM rescheduler needs to meet the maximum number of VMs changes rather than the average, as indicated by the green line in Figure 1. During the running time of a VM rescheduling algorithm, the VM scheduling algorithm is still processing incoming VM requests and some accomplished VMs may also be deleted. To mitigate the impact of VM allocation and deallocation, the VM rescheduling process is typically performed periodically (e.g., every day, mostly during off-peak hours). Figure 1 highlights the time (2:00 AM) to execute the VM rescheduling algorithm with a red point.

2.2 Problem Formulation and Motivation

2.2.1 Problem Formulation and Two Algorithms

In a data center, let $\mathcal{V}, \mathcal{P}$ be the set of VMs and PMs, respectively. On the supply side, a PM $i \in \mathcal{P}$ has two NUMAs[2], where NUMA j can provide $U_{i,j}$ CPU resources and $V_{i,j}$ memory resources. On the demand side, a VM $k \in \mathcal{V}$ requires u_k CPU resources and v_k memory resources and should be deployed on a single PM using $w_k \in \{1,2\}$ NUMAs. w_k is the number of NUMAs required by VM k (1 for single-NUMA deployment, 2 for double-NUMA). After deploying several VMs on PM $i \in \mathcal{P}$, it remains $\tilde{U}_{i,j}$ spare CPU resources on NUMA j. We define **X-core fragment** of PM i as $\sum_j (\tilde{U}_{i,j} \% X)$, i.e., the remaining CPU resources cannot be further utilized by any additional X-core VMs.

Given an initial assignment of M VMs each onto one of the N PMs, the VM rescheduling task is to reassign a subset of deployed VMs and migrate them each onto a new PM. The maximum number of VMs that we can migrate for a given task is called *Migration Number Limit (MNL)*. We formulate the VM rescheduling as an optimization problem that searches for a reassignment of MNL-selected VMs, in order to minimize the total X-core fragments across all PMs:

$$\text{Minimize:} \quad \sum_{i,j}\left(U_{i,j} - \sum_k \frac{x_{k,i,j}\cdot u_k}{w_k} - X y_{i,j}\right) \tag{1}$$

$$\text{Subject to:} \quad \sum_k \frac{x_{k,i,j}\cdot u_k}{w_k} + X y_{i,j} \le U_{i,j}, \tag{2}$$

$$\sum_k \frac{x_{k,i,j}\cdot v_k}{w_k} \le V_{i,j}, \tag{3}$$

$$\sum_{i,j} x_{k,i,j} = w_k, \tag{4}$$

$$\sum_k (1 - x_{k,i_k,j_k}) \le M, \tag{5}$$

$$x_{k,i,0} = x_{k,i,1}, \quad \forall k \in \{k | w_k = 2\}, \tag{6}$$

$$x_{k,i,j} \in \{0,1\} \text{ and } y_{i,j} \in \mathbb{Z}. \tag{7}$$

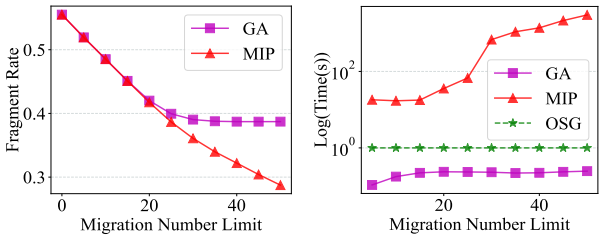
Here, $\{x, y\}$ are the decision variables, where $x_{k,i,j}$ represents whether VM k is deployed to the NUMA j of PM i in the new assignment (0 for No, 1 for Yes), and $y_{i,j}$ represents the maximum number of X-core VMs can be deployed on NUMA j of PM i using the remaining CPU resources. The objective in Equation 1 is to minimize the total X-core fragments.

Equation 2 and 3 enforce that the resource usage by VMs cannot exceed the total capacity of a PM. Equation 4 indicates that each VM must be deployed on exactly one PM. Equation 5, in which i_k and j_k are the initial PM id and NUMA id (0 for double-NUMA VMs) of VM k, means the total migration number should not exceed the limit. Lastly, Equation 6

[2] Non-uniform memory access.

Table 1: The VM types we consider in our experiments.

VM Types	large	xlarge	2xlarge	4xlarge	8xlarge	16xlarge	22xlarge
Requested CPU	2	4	8	16	32	64	88
Requested Memory (GB)	4	8	16	32	64	128	256
Deploy NUMA	Single	Single	Single	Single	Double	Double	Double
Percentage	11.01%	43.51%	16.13%	17.62%	6.22%	5.31%	0.20%



(a) Fragment Rate Performance. (b) Running time.

Figure 4: FR and Time of MIP and GA given different MNLs.

restricts VMs with double NUMAs from deploying both of its NUMAs on the same PM.

Note (1) each PM has two NUMAs; (2) w_k is a constant for each VM as determined by their types (Table 1). Thus, $\sum_{i,j} x_{k,i,j} = w_k$ (Equation 4) enforces that the actual NUMA allocation number of VM k matches the desired configuration. When $w_k = 1$, Equation 4 constraints VM k to be deployed on one NUMA of a PM; when $w_k = 2$, Equation 4 constraints VM k to be deployed on both NUMAs of a PM. Note that deploying VM k on two NUMAs of two different PMs (each PM hosting a NUMA) violates $x_{k,i,0} = x_{k,i,1}, \forall k \in \{k | w_k = 2\}$ (Equation 6). Because $w_k \neq 0$, it guarantees each VM is deployed.

Mixed Integer Programming (MIP) Solvers. The above optimization problem can be solved by an off-the-shelf MIP solver such as Gurobi [2] and CPLEX [1], which can find an optimal solution through algorithms of branch & bound, cutting planes, etc. In our experiments, we use the former. The primary issue of the MIP approach is that its poor time complexity prevents it from scaling to industry-scale data centers with thousands of PMs and VMs.

Greedy Algorithm (GA). To obtain a feasible solution within a short time frame, greedy algorithms are often used in industry data centers [3]. They normally include two stages: filtering and scoring. In the filtering stage, we calculate the change in FR for each VM if it is removed from its source PM, and only select the VM candidate that corresponds to the most significant drop in FR. In the scoring stage, we calculate the change in FR if the selected VM is migrated to each of the eligible PMs. We then greedily assign the selected VM to the PM that leads to the largest drop in FR. The above two stages are repeated until the MNL is reached. Although GA can significantly reduce the search space of the rescheduling problem, we show that it often fails to yield a good solution.

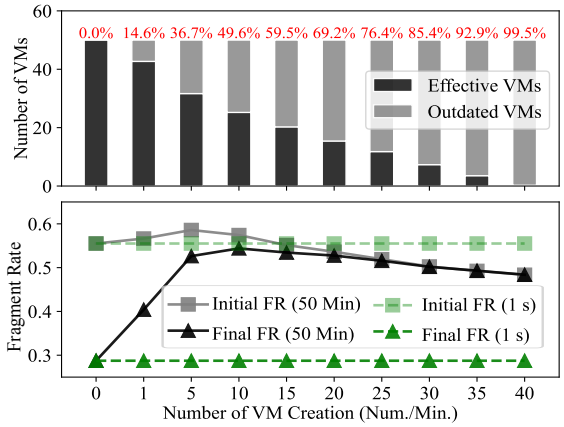


Figure 5: Effect of the Running Time of MIP at MNL 50.

2.2.2 Experiment Results

We conduct experiments to quantify the performance of the above MIP and GA algorithms in terms of FR and running time. We use a real dataset collected from a data center with 280 PMs and 2089 VMs. The detailed experiment settings will be found in Section 5.

Figure 4(a) depicts the FR performance of MIP and GA under different MNLs. MIP's FR is higher than the GA since it guarantees a near-optimal solution. Moreover, the FR gap between MIP and GA becomes larger as MNL increases, because GA only exploits the action that would lead to the most significant drop in FR when it migrates one VM.

In Figure 4(b), the time required for MIP and GA to generate a solution is displayed, and the green dash line indicates the one-second guarantee (OSG). If the solution time is less than 1 second, the FR performance is guaranteed and the impact of VM changes on FR is negligible. However, as the MNL increases from 25 to 50, the computation time of MIP grows exponentially, taking 1.78 minutes for 25 migrations and 50.55 minutes for 50 migrations. This poor running time performance is unacceptable in data centers where new VM requests continually come in, and the problem state is constantly varying. In contrast, the running time of the GA algorithm is lower than the OSG and remains constant when the MNL exceeds 25. After migrating 25 VMs, the GA algorithm can no longer find any more VMs that can lower the FR.

The dynamic nature of VM allocation and deallocation can affect FR performance if the migration plan is based on the current VM-PM mapping and solved after 50 minutes. We

quantify the performance reduction from two aspects. First, we measure the percentage of failed VMs during the VM Rescheduling stage, where a VM is considered a failure if it cannot be found, or there are not enough available resources to migrate it from the source to the destination PM. Second, we measure the reduction in FR percentage, which is calculated by comparing the initial FR and final FR after migrating all 50 VMs using MIP's solution. The top subfigure in Figure 5 shows the failed VM percentage, and the bottom subfigure shows the FR reduction percentage.

We can see from the figures that the VM Rescheduling can successfully schedule all 50 VMs, and the FR performance is 93.14% between the initial and final FR after migrating all the 50 VMs at VM generation speed 0/min, meaning no VM changes. However, the percentage of failed VMs increases, and the FR performance decreases as the VM changes speed rises. If the VM changes speed exceeds 40/min, there is no successful VM migration, and no performance gain, rendering the migration plan useless. The green dash lines show the initial and final FR if the MIP can run in 1s, we can see that the dynamic of VM allocation and deallocation has no effect on the FR performance. So, the running time of VM rescheduling algorithm should meet the one-second guarantee.

Limitations of the Current Methods. From the above experiment results, we see that the MIP solver has the scalability challenge due to its intensive computation overhead. Therefore, the greedy algorithm is currently used in data centers to achieve fast FR decrease. However, its FR performance can be stuck into a local optimal since it is focused on one VM at each step, without considering future steps. To reduce the execution time of MIP solvers, some current methods rely on estimating feasible solutions using proprietary heuristic methods and then using branch-and-cut techniques [19, 43] to identify optimal solutions. The hand-tuned heuristics are based on human expertise to overcome the scalability challenge. The heuristics aim to prune the search space to make the problem tractable. Yet, they all rely on human expertise and operational experience. It is a tedious, time-consuming iterative process even for experts. Even worse, such a process has to be done and tuned for every VM rescheduling problem as there are no universal heuristics for all scenarios.

3 Design of VMR^{2}L

In this section, we describe in detail the design of VMR^{2}L, including overview and design challenges (§ 3.1), the formulation VM rescheduling as an RL problem (§ 3.2), the neural network architecture of the RL agent (§ 3.3). We refer the reader to Appendix A.1 for a brief overview about (deep) RL and Appendix A.4 for details on RL training.

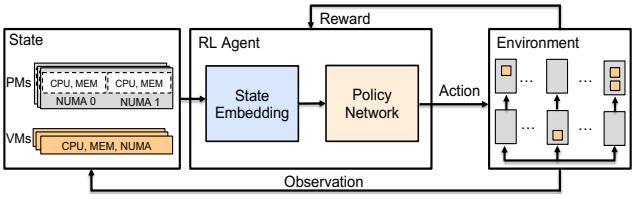


Figure 6: The overall VM rescheduling framework.

3.1 VMR^{2}L Overview

VMR^{2}L is represented by an agent that utilizes a neural network to make decisions. When a VM rescheduling event occurs, such as when there are no more PM resources left for a new VM request, the agent receives the current state of the data center as input and generates a rescheduling action. The state captures the status of all PMs and VMs, while the actions specify the migration steps required to move a VM from its source PM to a destination PM.

VMR^{2}L trains its neural network via offline simulation experiments. The neural network is trained end to end in an episode setting. In each episode, VMR^{2}L attempts to reschedule a VM from its source PM to destination PM, observes the outcome and receives a reward from the environment after each action. The reward is based on VMR^{2}L's rescheduling objective, such as minimizing fragment rate. The RL algorithm uses this reward signal to gradually improve the rescheduling policy. The framework of VMR^{2}L (Figure 6) is general and adaptable to different objectives. We illustrate in our experiments how the same design can be used for various workloads and goals, such as minimizing the number of migrations while meeting a specific FR objective.

3.2 VM Rescheduling as an RL Problem

Under the Markov Decision Process (MDP) framework, at each time step t, the scheduling agent selects an action $A_t = a$, given the current state $S_t = s$, based on its policy π_θ. More specifically, the agent's action is to select a VM and a PM, by which it reschedules the selected VM from its source PM to the chosen destination PM. T is the state transition function defined as $T : S \times A \to S$, which maps the current state to the next state given an action. The episode terminates if the selected VM cannot find a suitable PM or the number of actions taken reaches a pre-defined threshold. The latter is consistent with the requirement upon deployment that each rescheduling comes at a cost and we may only move the VMs around for a limited number of steps. Notably, in this problem, T is deterministic as the environment has no aleatoric uncertainties. In other words, we can directly simulate the next state and the change in FR given the current state and action taken by the agent. Thus, we build a cheap simulator so that our agent learns by interacting with the simulator instead of with the real environment. Given the above design, we now formalize the state, action, and reward of VMR^{2}L 's RL framework.

State Representation. The state is what the DRL agent takes as input at each rescheduling step. The state representation includes **1) PM features** - 8 in total: the first four features are the remaining CPU resources on NUMA 0 and NUMA 1, as well as the remaining memory resources on NUMA 0 and NUMA 1. The other features are manually designed, which are the FR and the fragment size of each NUMA. The last four features allow the agent to directly access the internal fragment information. **2) VM features** - 14 in total: the first four features are the requested CPU and memory resources of NUMA 0 and NUMA 1, and the last two are designed to be the fragment sizes of NUMA 0 and NUMA 1. We also include the 8 features from its source PM to better incorporate the locality information of each VM. Note that if a single NUMA is requested, we use zeros as placeholders for the other NUMA. For training efficiency, we scale each feature dimension with min-max normalization.

Action Representation. The DRL agent interacts with the environment via actions. The action at each step can be represented as a 2-tuple (k, i), where $k \in \mathcal{V}$ and $i \in \mathcal{P}$. Specifically, the action is to reschedule a VM k from its source PM to a destination PM i. Note that the source PM can be retrieved once we select k, so we do not include it in the action space.

Reward Representation. Reward represents the immediate evaluation of the rescheduling effect for each action under a certain state. The goal of VM rescheduling is to minimize the FR across all PMs. While in principle we could return the FR of all PMs as a single final reward to the agent after finishing an entire episode, it corresponds to a form of sparse reward and it is known to be difficult for training [53], as the action at each step only contributes marginally to the change in the overall FR. Instead, we propose to generate dense rewards and use the change in FR on the source PM and the destination PM as an intermediate reward at each step. As such, the range of the reward is naturally scaled down to a $[-2, 2]$ range, which we further normalize by dividing with a constant c (c=4 in our case)[3]. Equation 8 calculates the fragment size on each NUMA, and Equation 9 shows fragment changes for the source PM and destination PM.

$$S_i = \sum_{j=0}^{1} \left(\tilde{U}_{i,j} \% X\right) \div c, \tag{8}$$

$$R = \left(S_{\text{before, src}} - S_{\text{after, src}}\right) + \left(S_{\text{before, dest}} - S_{\text{after, dest}}\right), \tag{9}$$

where $S_{\text{before},\cdot}$ and $S_{\text{after},\cdot}$ show the fragment changes before and after this selected VM leaves (enters) the source (destination) PM. Recall that we focus on $X = 16$ in this paper.

[3]It is a common practice to use reward scaling to get better results for deep RL [28].

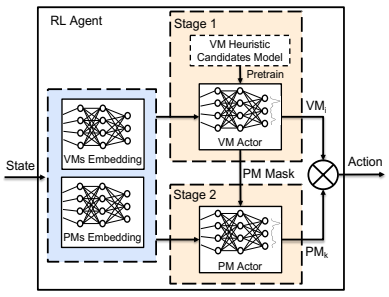


Figure 7: Two-stage RL Agent.

3.3 Neural Network Architecture

3.3.1 Two-Stage Agent with Knowledge Distillation

Two-Stage Framework. When the RL agent takes action (k, i), meaning to reschedule a VM k from its source PM to a destination PM i, PM i might not have enough available CPU or memory to host VM k. In addition to resource constraints, in practical scenarios, we must consider additional constraints to ensure service stability. For example, a critical application might request for several VMs to be hosted across different PMs, which can be enforced in the form of a hard anti-affinity constraint.

Off-the-shelf RL models impose such hard constraints typically by invoking a heavy penalty if an illegal action is chosen, or by directly setting the probabilities for all illegal actions to be zero. As we show Section 5.3, heavy penalties can result in gradient instability and thus lead to an inferior convergence rate. On the other hand, as the size of the action space is $O(|\mathcal{V}| \cdot |\mathcal{P}|)$, masking all illegal actions is overly time-consuming and fails to meet the latency requirement.

To better accommodate a variety of constraints, we leverage the characteristics of the VM rescheduling problem and design a two-stage framework that allows the action tuple to be generated sequentially. As shown in Figure 7, in Stage 1, the VM actor selects the VM candidate to be rescheduled. Once a candidate VM is selected, we can efficiently mask out all the PMs that cannot host the candidate VM and then proceed to Stage 2, where the PM actor selects an appropriate destination PM from the remaining PMs. Additionally, when we select a VM to reschedule, a considerable portion of the PMs cannot meet its resource requirements. The exploration of RL agent on these PMs inevitably hinders the learning process. The proposed framework can effectively reduce the large action space and thus mitigate the exploration challenge.

Knowledge Distillation. The aforementioned two-stage RL Agent framework helps to narrow down the search space of the PM actor. However, the main bottleneck of the large action space originates from the number of VMs as there are always more VMs than PMs in data centers. Furthermore, the VM actor cannot obtain a good reward unless the PM actor also selects an appropriate destination PM.

To make the most out of two sequential actors, we address this challenge by injecting domain knowledge from a heuris-

tic VM candidates model, such as the greedy algorithm or a model with manually engineered features designed by a domain expert, to pretrain the VM actor via knowledge distillation [30, 41]. Specifically, before the heuristic algorithm selects a VM to be rescheduled, it first needs to generate a score for each VM. The scores can be viewed as samples from a soft target distribution, which are commonly referred to as soft labels [60]. To transfer the rich domain knowledge to VMR^{2}L, we pretrain the VM actor to learn the mapping from states to the soft labels for each VM. The detailed procedures for offline pre-training are outlined in Algorithm 2 in Appendix[4]. In this way, knowledge distillation allows domain experts to directly inject knowledge into the model. The VM candidate's agent can exhibit preliminary maturity at the onset of its learning phase, and can readily produce a positive reward once the PM actor explores a good action. Once the VM actor is put into the training environment, we continue to improve its policy end-to-end together with the PM actor.

3.3.2 Feature Extraction with Sparse Attention

Scalability. To make effective rescheduling decisions, VMR^{2}L must extract meaningful representations of the state observation, which include features of each individual PM and VM as well as their affiliations. As Figure 1 implies, the number of VMs can vary drastically even in the same data center. In fact, at **anonymous** company's in-house data center, the number of VMs varies during the day. This implies that the size of the features at each time step is also highly dynamic. To encode these features, one option is to concatenate features from all VMs and PMs and flatten them at once. However, this approach cannot handle an arbitrary number of VMs as neural networks usually require fixed-sized inputs, and it also requires a model with a large number of parameters that would be difficult to train.

Instead, we propose to share two small embedding networks across all VMs and PMs — one to process each PM's features and another one to process each VM's features. As such, the number of weight parameters is *independent* of the number of machines in the system. This is achieved via an attention-based transformer models [64] but tailored for rescheduling. Transformers have demonstrated strong performances in Natural Language Processing [15, 50, 51], Computer Vision [17], as well as combinatorial optimization, such as in vector bin-packing [36, 67]. However, compared to bin-packing, there is a notable difference in VM rescheduling: we must choose from a set of VMs that have already been assigned to PMs.

Tree-level Features. Consider a PM with 2 CPUs left. It contains a VM with 4 CPUs and another VM with 2 CPUs. Suppose a second PM has a fragment size of 8 while hosting

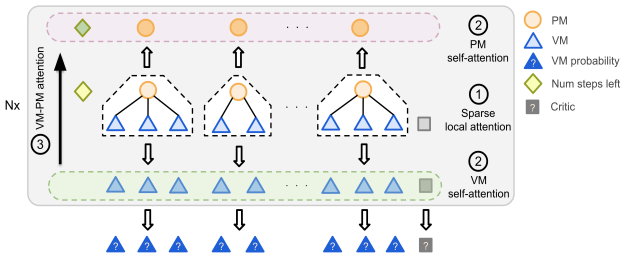


Figure 8: VM actor architecture with sparse local-attention to capture the tree-level features.

a VM with 8 CPUs. In order to minimize the total 16-core fragments, an ideal approach would be to first remove the two VMs with 2 and 4 CPUs from the first PM, and then reassign the VM with 8 CPUs from the second PM to the first PM. However, if we merely include the source PM's features in each of the VM's features and feed them into the vanilla transformer model, there will not be sufficient information for the two actors to take the above actions. Instead, each VM must also be aware of the other VMs that are hosted on the same PM, which is not possible in the vanilla transformer model. In fact, each PM can be viewed as a tree of depth one, where the PM acts as the root node and the VMs it hosts act as the leaf nodes. In order to allow every VM to recognize which other VMs are hosted on the same PM, we propose to include an additional stage of *sparse local-attention* within each PM tree, i.e., we only allow PMs and VMs to attend to each other if and only if they belong to the same tree.

Architecture Overview. To better incorporate the proposed sparse local-attention module, we modify the vanilla transformer architecture as follows[5]. The model is composed of several attention blocks, where each block includes three stages as shown in Figure 8:

1. All PMs and VMs exchange information if they belong to the same tree via sparse local-attention.
2. Each PM attends to other PMs' updated embeddings and each VM attends to other VMs' updated embeddings with self-attention.
3. The new VM embeddings are allowed to attend to the new PM embeddings through VM-PM attention.

At the end of the three stages, each machine is then further processed by two point-wise dense layers with ReLU activation and layer norm [6]. The updated embeddings for each machine is then feed into the next block and the process repeats. Finally, the VM embeddings from the final block are linearly projected into a set of logits followed by the Softmax operation [8] to generate the probability of each selected VM.

As for the PM actor architecture, we adapt the vanilla transformer encoder-decoder module, since we can directly inject the graph information by including the VM and PM embeddings from the VM actor as input. We only feed in the selected

[4]Note that the PM teacher can also be replaced with a heuristic model designed by a field expert.

[5]A more rigorous formulation of the proposed feature extraction module can be found in Appendix A.5.

VM candidate to the encoder, while the decoder still takes in all the PMs. Additionally, we also add the VM-PM attention score from stage 3 for the selected VM, since the score implies why the VM actor selects the chosen VM and which PMs it attends to in the process. The output embeddings of each PM is linearly projected into a logit. Based on the selected VM, we mask out all the illegible PMs by setting their logits to be $-\infty$. The remaining logits are translated into the probability of selecting each PM as the destination PM.

3.3.3 Risk-seeking Evaluation

Indeed, our experimental results have underscored the efficacy of VMR^{2}L. Therefore, particularly during non-peak periods when rescheduling requests are relatively fewer, we can prioritize higher accuracy (lower FR) over latency requirements. The VM rescheduling problem is distinct from general RL problems as it allows access to a perfect world model without aleatoric uncertainties. In other words, the simulator can directly compute the final state and corresponding FR for a given initial state and sequence of actions. As a result, a viable strategy, which we refer to as *risk-seeking evaluation*, is to **sample multiple trajectories during policy evaluation (inference time), deploying the one with the highest reward**. In Appendix A.2.1, we provide a theoretical perspective on the number of trajectories to sample during deployment. It is worth mentioning that the concept of utilizing the best-performing trajectory can be further extended to the training process, known as *risk-seeking training* [47]. We discuss this in greater detail in Appendix A.3 and designate this extension as future work.

Action Thresholding. To obtain varied trajectories during inference, we sample actions from the learned policy, $\pi(\cdot|s)$, rather than exclusively selecting the action with the highest probability. It is important to note that the learned policy is likely to differ from the optimal policy due to approximation errors. In Appendix A.2.2, we demonstrate that **actions with low probabilities are likely to be suboptimal**. Although these actions may not be executed with high probability in a single period, they are likely to be performed throughout the entire trajectory. This is because the probability of an event not occurring in independent trials decreases exponentially with respect to the number of trials. Therefore, in our implementation, we **mask out actions assigned significantly low probabilities**. Figure 18 illustrates that trajectories with lower FR are more likely to emerge when low probability actions are masked out.

4 Implementation

RL algorithm and Neural network architecture We implement VMR^{2}L based on the CleanRL framework [32] using PPO as the backbone [57]. VMR^{2}L contains about 8.5K lines of Python code. The overall framework is implemented using PyTorch [21]. As the model can be trained end-to-end, it does not require manual feature engineering and can learn statistically what is important from the features. On the other hand, domain experts can inject prior knowledge into VMR^{2}L in the form of heuristic models through knowledge distillation. Additionally, the number of model parameters is independent of the number of VMs or PMs, allowing it to scale to large data centers. We list the hyperparameters in Appendix A.9.5. We also built a visualization tool to allow end-users to better interpret VMR^{2}L. Some sample trajectories are included in Appendix A.10.

VM Rescheduling Simulator. While DRL can be very powerful, its main drawback is the amount of training data required [39]. In light of this, we design a simulator for the VM rescheduling task. The simulator follows the OpenAI Gym environments [9] including specific file hierarchy and function abstractions. Given an existing VM-PM mapping and a rescheduling action, we can directly simulate the change in FR caused by the action. Thus, during training, VMR^{2}L only needs to interact with the simulator instead of with the real environment, which drastically lowers the amount of real-world data required to train the agent. In Appendix A.7, we validate the fidelity of our simulator by comparing the FR changes under the MIP's migration plan.

Experiment Setup. Experiments are done on a Linux server with 8 CPUs (Intel Xeon E5) and 1 GPU (NVIDIA RTX 3090). The training time for knowledge distillation, VMR^{2}L with sparse attention, and VMR^{2}L without sparse attention is 2h, 92h, and 48h, respectively. We sample 8 trajectories for VMR^{2}L and report the best result. We report the average over 3-5 runs with different random seeds.

5 Evaluation

We conduct extensive experiments on the datasets introduced in Section 4 to answer:

- How far is VMR^{2}L from the optimal solution? (§ 5.2)
- How much performance gain does each design component of VMR^{2}L provide? (§ 5.3)
- How well does VMR^{2}L generalize and scale? (§ 5.4)

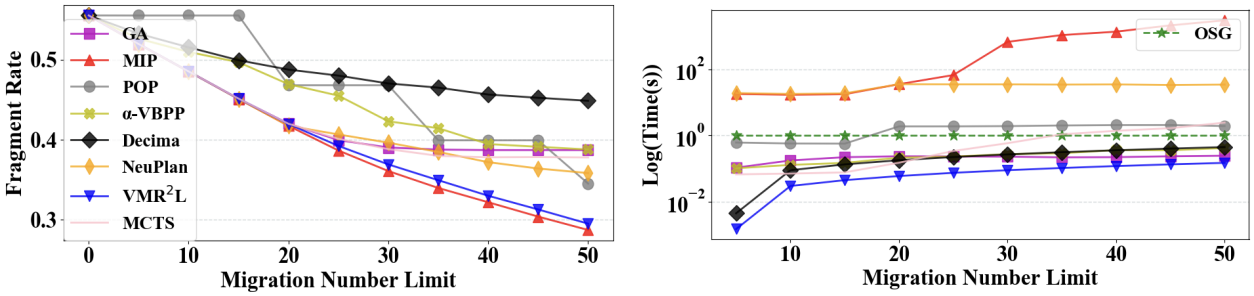
5.1 Existing Baseline Algorithms

As later discussed in Section 6, existing baselines can be summarized into six categories: heuristic algorithms (e.g., greedy, α-VBPP), optimization algorithms (e.g., MIP), approximate algorithms (e.g., POP), search-based algorithms (e.g., MCTS), deep learning-based methods (e.g., Decima), and combination-based methods (e.g., NeuPlan). For each type, we implemented at least one algorithm to compare.

MIP Algorithm: as introduced in Section 2.2.1.

Greedy Algorithm: as introduced in Section 2.2.1.

Vector Bin Packing Problem (α-VBPP): We generalize the VBPP [44] algorithm for initial scheduling to rescheduling. We first divide the entire episode into MNL/α stages. During each stage, we greedily remove α number of VMs that lead



(a) Fragment Rate Performance

(b) Inference Time

Figure 9: Fragment rate and running time of VMR^{2}L compared with the baselines at different Migration Number Limits.

to the most fragments, and then apply VBPP to treat them as incoming VMs. We carefully tune α (10 in our case) to achieve the best FR reduction.

Partitioned Optimization Problems (POP): The POP method [42] solves the optimization problem formulated in Section 2.2.1 by splitting the large-scale problem into smaller subproblems (each contain a subset of VMs and PMs in the data center) and coalescing the resulting sub-rescheduling solution into a global rescheduling solution for all VMs.

Monte-Carlo Tree Search (MCTS) [70]: As traditional search based methods need to perform multiple rollouts during inference time to achieve a good performance, we use DDTS [70] to prune the search space.

Decima [40]: Decima uses a graph neural network to encode the VM and PM information and trains using deep RL. Decima balances the size of the action space and the number of actions required by decomposing VM rescheduling decisions into a two-dimensional action, which outputs (i) the VM that needs to migrate, and (ii) an upper number of PM subsets to choose as the destination.

NeuPlan [69]: NeuPlan uses a two-stage hybrid approach to address MIP's scalability issue. In the first stage, an RL agent takes in the problem as a graph and generates the first few actions to prune the MNL search space. In the second stage, it uses a MIP solver for the remaining MNLs. A relax factor β (30 in our case) is used to control the size of the MNL space to explore by MIP.

5.2 Overall Performance

Figure 9 shows the FR and running time of all the six methods. The absolute evaluation numbers for Figure 9 can be found in Appendix A.9.6.

Fragment Rate. The experiment results in Figure 9(a) reveal that VMR^{2}L can achieve +14.48%, +17.60%, +22.37%, +23.71%, 23.84%, and +34.17% performance gain when compared to POP, NeuPlan, MCTS, GA, α-VBPP and Decima respectively, when the task is to perform 50 reschedulings on the medium dataset. Notably, VMR^{2}L is merely 2.67% behind the optimal MIP solution (0.2953 vs. 0.2859).

It is noteworthy that the performance gap between VMR^{2}L and optimal MIP does not increase with the increase in the MNL. Therefore, it can be concluded that VMR^{2}L's performance is relatively stable, and the difference with optimal MIP remains consistent even at higher MNL values.

Although α-VBPP can reduce FR with more MNL, its performance is inferior to that of GA. This is because α-VBPP only removes the α worst VMs for each stage based on a single timestep, failing to consider future opportunities to replace them back and achieve even higher FR reduction. POP fails to achieve a good performance since it still relies on MIPs to solve each subproblem. To meet the second-level latency requirement, we must divide the problem into many subproblems, causing its solutions to be only locally optimal. On the other hand, Decima reduces the huge action space by limiting the PM actor to only select from a subset of PMs, but the subsampling of PMs is completely random, as opposed to our solution. While MCTS with DDTS uses neural networks to prune the search space, it still requires a significant number of rollouts to achieve stable performance. Lastly, NeuPlan is able to achieve a low FR, since it solves a large subproblem entirely with MIP. We implement NeuPlan as follows: if MNL is less than 20 steps, MIP is used to solve the optimization problem. If MNL is larger than 20, the first 20 VMs are migrated by MIP and the remaining VMs will be handled by RL. Notice that NeuPlan's FR is higher than VMR^{2}L after MNL = 20, since after this step NeuPlan relies entirely on its RL agent, which fails to learn a good policy given such a huge state and action space. The detailed fragment rate on 200 individual Datasets are shown in A.6.

Inference Time. From Figure 9(b), we can see that the solution time of GA, α-VBPP, Decima and VMR^{2}L is less than OSG. VMR^{2}L can generate one trajectory within 0.15s when MNL = 50. In comparison, MIP requires 50.55 minutes to provide the optimal solution. The running time of POP can be adjusted by setting how many subproblems to divide into. To meet the latency requirement of the VM rescheduling task, we set this number to 16 so POP can deliver a solution within 1.94s. Both GA and α-VBPP are greedy algorithms and can provide the solution within 1s. Decima requires 0.45s, which

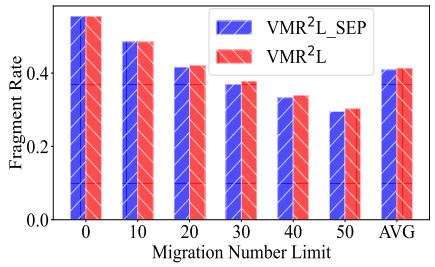


Figure 10: Fragment Rate Gap between VMR^{2}L and VMR^{2}L _SEP.

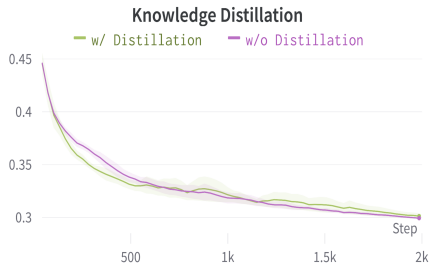


Figure 11: Convergence rate w/ and w/o distillation.

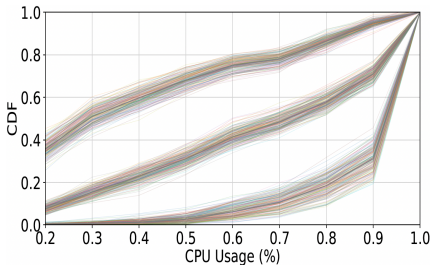


Figure 12: CPU Usage on PMs under Different Workloads.

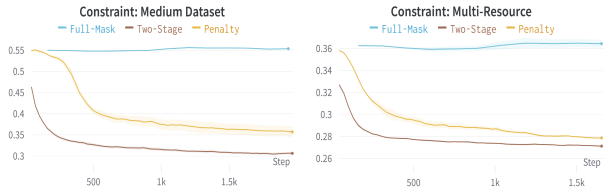


Figure 13: Performances under Various Constraints.

Aff. Level	0	1	2	3	4	8
Aff. Ratio	0	1.12%	1.86%	3.46%	6.50%	38.3%
FR	0.3032	0.3029	0.3034	0.3048	0.3045	0.3306
MIP	0.2859	0.2860	0.2860	0.2862	0.2864	OOT

Table 2: FR under Different Affinity Constraint Levels.

is at a roughly same scale as VMR^{2}L, since both use end-to-end deep RL. Lastly, NeuPlan takes 34.8s to yield the final solution since it still needs MIP to solve 20 steps.

5.3 VMR^{2}L Deep Dive

We break down the impact of our key ideas and techniques on VMR^{2}L's performance.

Two-Stage: Different Service Constraints. To better analyze the proposed two-stage framework, we compare with two baselines: **i) Penalty**: a penalty of -5 is given if the agent takes an illegal action, and **ii) Full-Mask**: we generate the VM candidate and the PM destination simultaneously and set all illegal pairs to have probabilities of zero.

As the original PM and VM types in Table 1 are CPU-bounded [6], we consider traces from an additional smaller data center that has more complicated multi-dimensional resource constraints with more VM and PM types. Additionally, we also consider a practical service constraint in the form of a hard anti-affinity, where a VM cannot be placed on the same PM with some other selected VMs. To enforce anti-affinity, we can mask out all the PMs that host conflicting VMs in stage 2 after selecting a VM candidate in stage 1. Detailed experiment settings and additional plots can be found in Appendix A.9.2.

As we can see from Figure 13(a) and 13(b), **Penalty** suffers from a slower convergence to a sub-optimal level, since the large negative penalties required to eliminate illegal actions tend to dominate the gradient signal especially during early stages of training. **Full-Mask** does not converge under both constraint settings, since its action space is the product of the number of VMs and the number of PMs, which leads to poor exploration. Additionally, it requires 4.5s to generate 50 rescheduling steps, deeming it infeasible for low-latency applications[7]. In comparison, **Two-Stage** decomposes the action space by designating stage 1 to focus on the set of VM candidates and stage 2 to focus on the set of PM destinations, resulting in much better exploration.

As for service anti-affinity, we typically observe an affinity ratio requirement to be under 5% in real-world traces. Affinity ratio means the average percentage of VMs that a given VM conflicts with. In Table 5.3, we see that VMR^{2}L maintains consistent performance under typical levels of anti-affinity constraints. To demonstrate the robustness of VMR^{2}L to extreme constraint levels, we further evaluate it when the affinity ratio surges to 38.3% and see that VMR^{2}L is still able to achieve a reasonable FR.

Speedup via Knowledge Distillation. Figure 5.3 shows VMR^{2}L's learning curve with and without imitation learning (IL). From this Figure, we can see that both VMR^{2}L and VMR^{2}L without IL can converge to the same return within 500 training steps. VMR^{2}L without IL can converge at 400 steps while VMR^{2}L converges at 200 training steps which are two times faster than VMR^{2}L without IL. For our VM rescheduling task, it usually takes more than 48 hours to train the VMR^{2}L agent to converge due to the huge state and action space. As such, VMR^{2}L with imitation learning can greatly improve the training speed.

Number of Trajectories to Sample. During the deployment stage, VMR^{2}L will select different VM and PM given the same vm-pm mapping observation since both VM and PM are sampled from probability distributions. We can run the designed VM simulator multiple times to select the best ac-

[6] CPU-bounded means that a PM must run out of CPU before running out of memory

[7] Due to its significantly longer inference time, we evaluate **Full-Mask** less frequently.

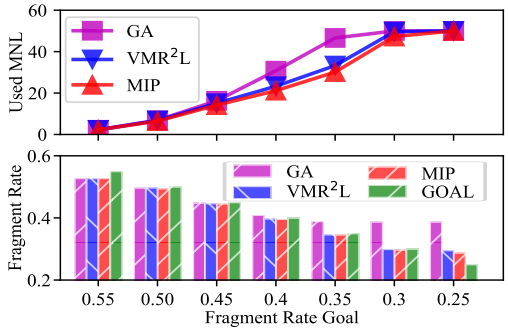
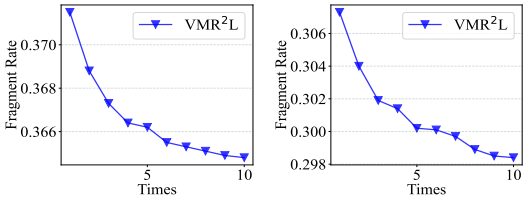


Figure 14: MNL Performance under Different FR Goals.



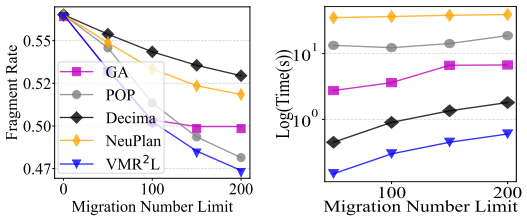
(a) MNL=30 (b) MNL=50

Figure 15: Fragment rate at different numbers of trajectories.

tion combination. However, we need to balance the trade-off between fragment rate reduction and inference time.

Figure 15 shows the fragment rate decreases when the running times increase for both experiments of MNL 30 and 50. However, the fragment rate reduction becomes slow after $t = 6$ and $t = 8$ for MNL 30 and 50 respectively. We leverage the elbow method to select $t = 8$ for VMR^{2}L, and the fragment rate performance can be averagely improved by 1.72% and 2.73% at MNL 30 and 50 compared with the case that the designed VM simulator is only executed once.

Different Service Objectives. VMR^{2}L's flexibility enables it to learn different policies depending on the high-level objective. We now consider a new objective: given FR goals, we would like to minimize the MNL to reduce migration costs. As seen in Figure 14, the top subfigure displays the used MNL, while the bottom subfigure shows the FR, both sharing the x-axis with the FR goals. The used MNL of GA, VMR^{2}L, and MIP increase with higher FR goals (lower FR). At a goal of 0.55, the average FR of the dataset is 0.53, resulting in FR lower than the goal with the same MNL. MIP and VMR^{2}L achieve 14.77% and 11.11% less MNL than GA, respectively. VMR^{2}L performs similarly to MIP in terms of MNL performance, with a difference of only 3.66%, but with millisecond-level solution time. On the other hand, GA is stuck at goal 0.4 due to its suboptimal performance. None of the three methods can achieve the FR goal of 0.25, since the optimal FR is 0.2859 at MNL 50.



(a) Fragment Rate (b) Running Time

Figure 16: Fragment rate and time performance between GA and VMR^{2}L on the large dataset.

5.4 Generalization and Scalability of VMR^{2}L

Generalizing to Different MNLs. In practice, the required migration number limit (MNL) can constantly vary due to changing business requirements such as target fragment rates. We show that we can readily achieve good performance by only training one VMR^{2}L agent with MNL = 50, and deploying it for MNLs $= \{10, 20, 30, 40, 50\}$. For comparison, we train a separate VMR^{2}L agent for each MNL, which we denote as VMR^{2}L $_{\text{SEP}}$. As shown in Figure 10, VMR^{2}L performs only marginally worse than VMR^{2}L $_{\text{SEP}}$ with an average FR performance gap of 1.16%. This suggests that the VMR^{2}L agent trained with a large MNL can be readily applied to the tasks with smaller MNLs. It avoids the overhead of maintaining a separate VMR^{2}L agent for each MNL.

Generalizing to Abnormal Workloads. It is well-known that common deep RL applications tend to suffer from performance degradation as a result of distribution shifts. [35]. However, within actual service traces, there are often anomalous periods during which workloads (the percentage of available CPUs on PMs) deviate from the typically observed workloads during festivals and promotional events. To determine when VMR^{2}L can generalize to abnormal levels of workload, we additionally collect two datasets with *Low*(L) and *Medium*(M) workloads. The medium dataset acts as *High*(H) workloads. The workload distributions of each training file from the three datasets are shown in Figure 12. Note that the three datasets have strictly non-overlapping workload distributions, i.e., we cannot find a training sample from *High* that has workload similar to *Medium* or *Low*.

We train VMR^{2}L on one or mixed workload datasets and evaluate the FR on each individual workload level. The results are summarized in Table 5.4. VMR^{2}L (L,H) means that we train VMR^{2}L on both *Low* and *High* datasets. As MIP runs out of time due to the larger MNL used on L and M, we choose POP as the standard baseline since it is easy to tune and exhibits strong FR performances. We see that VMR^{2}L performs well when trained on a workload level similar to the test workload. When VMR^{2}L trains with workloads smaller than the test workload, VMR^{2}L suffers from performance degradation. Intuitively, a high workload requires the agent to create available resources by first removing existing VMs away from the destination PM, but this action is less common

Methods	L (MNL=100)	M (MNL=100)	H (MNL=50)
GA	0.256(-2.7%)	0.276(-8.0%)	0.387(-10.9%)
VMR^{2}L (L)	**0.237(+4.8%)**	0.261(-2.7%)	0.424(-18.6%)
VMR^{2}L (M)	0.239(+4.0%)	0.238(+6.3%)	0.422(-18.2%)
VMR^{2}L (H)	0.243(+2.4%)	0.248(+2.4%)	**0.303(+12.2%)**
VMR^{2}L (L,H)	**0.237(+4.8%)**	**0.237(+6.7%)**	0.326(+5.5%)
POP	0.249	0.254	0.345

Table 3: Generalization to Different Workloads.

and thus cannot be effectively learned on lower workloads.

Remarkably, when training with both L and H, VMR^{2}L can learn a general policy and perform the best on M without ever experiencing medium workloads. Thus, we suggest end-users of VMR^{2}L to include samples with both high and low workloads from their data centers for training, and VMR^{2}L can effectively close the gaps in the train data and generalize better when observing abnormal workloads.

Scalability to the Large Dataset. We conduct experiments on a large dataset with 4546 VMs and 1176 PMs to analyze the scalability of VMR^{2}L. Figure 16 shows the FR and time performance of different baseline methods with the MNL from 50 to 200. The MIP is not included in this experiment since we cannot get a solution within 50 minutes. As it turns out, VMR^{2}L again achieves better performance than the baselines on both FR and solution time.

From Figure 16(a), we can see that VMR^{2}L can achieve average 2.01%, 2.08%, 6.14% and 7.87% and max 2.34%, 5.15%, 8.6%, and 10.5% performance gain compared with POP, GA, NeuPlan, and Decima respectively. Decima and NeuPlan both cannot achieve good FR performance on such a large dataset. NeuPlan mostly relies on RL's solution since MIP cannot work with MNL larger than 20. GA gradually stops reducing FR after 100 steps. POP and VMR^{2}L continue to reduce FR even at MNL 200. POP achieves worse FR than GA before 100 and better FRs for larger MNLs.

Figure 16(b) shows the running time to generate a migration solution with MNL set from 50 to 200. GA, POP, Decima, NeuPlan, and VMR^{2}L can generate a solution within 4.91s, 14.53s, 1.125s, 37.23s, and 0.375s on average. GA increases the time when MNL is less than 150. GA calculates the score of all the VMs and PMs as a metric to evaluate their migration value. The calculation time increases with the number of VMs and PMs. POP costs more time with a bigger MNL. VMR^{2}L increases time linearly from MNL 50 to 200. For VMR^{2}L, the neural network inference time will only increase minimally with the number of PMs and VMs. Its inference time is mostly affected by MNL and trajectories number 5.3. Decima needs three times than VMR^{2}L since the GNN needs to encode the VM-PM information. NeuPlan needs MIP to solve 20 MNL.

6 Related Work

Connections to Bin Packing. The use of optimized placement mechanisms proved to be successful in a broad set of use cases, including production quality scenarios [4] as well as transportation logistics [10, 18, 31, 66]. A typical solution exploits heuristics based on bin packing [44]. In fact, VM placement can be modeled as a bin-packing problem, where VMs and PMs are objects and bins, respectively. Bin packing typically involves packing a set of items into fixed-sized bins such that the number of bins required [10] or the total surface area is minimized [18, 31]. However, there are two notable differences. First, the problem of VM rescheduling concerns adjusting an initial assignment of VMs to PMs and has practical value in the industry, since some VMs might terminate causing the problem state to change. On the other hand, rebalancing items that are already packed in bins has received little attention in the context of other traditional bin-packing applications. Second, the total number of VMs and PMs in a data center can easily go into the range of several thousands or more [66], and is far more than the typical scale of bin packing problems, which typically involve no more than a few hundred items [14, 37, 68, 70].

RL for Optimization Problems. RL has been recently introduced to solve optimization problems, e.g., building ML compilers and optimizing neural network architectures [26]. In particular, RL is used to select branching variables or find cutting planes in the Branch-and-cut method [11, 20, 22, 23, 54, 59, 61]. Besides, RL can also be applied to existing heuristics for MIPs to further increase the quality of solutions [7, 49, 58]. However, the above approaches are not directly appropriate for the VM rescheduling task due to their poor computation complexity. Although they are designed to accelerate MIPs, as shown in Section 5.2 even a state-of-the-art technique as POP [42] fails to deliver a satisfying solution within the second-level time limits of the VM rescheduling task. VMR^{2}L makes two unique contributions to solving the VM rescheduling problem. First, it uses two actors framework to efficiently solve the large action space. Second, it ingests the domain knowledge from a VM heuristic candidates model to pretrain the VM Candidate's actor.

7 Conclusion

We present VMR^{2}L, a deep RL approach to solve the VM rescheduling problem. VMR^{2}L uses a two-stage framework to address the huge action space challenge. It also incorporates a sparse attention module to better capture the local graph information, which is critical for VM rescheduling. Lastly, we recognize the VM rescheduling task as a special RL problem where the goal is to optimize for the best-case performance instead of on-average. To this end, we further design several risk-seeking components to facilitate effective decision-making. Extensive experiments on two datasets demonstrate the effectiveness of VMR^{2}L.

References

[1] Cplex optimizer. https://www.ibm.com/analytics/cplex-optimizer.

[2] Gurobi solver. https://www.gurobi.com/.

[3] Kubernetes scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[4] AHMAD, R. W., GANI, A., HAMID, S. H. A., SHIRAZ, M., YOUSAFZAI, A., AND XIA, F. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of network and computer applications 52* (2015), 11–25.

[5] ARULKUMARAN, K., DEISENROTH, M. P., BRUNDAGE, M., AND BHARATH, A. A. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine 34*, 6 (2017), 26–38.

[6] BA, J. L., KIROS, J. R., AND HINTON, G. E. Layer normalization, 2016.

[7] BARRETT, T., CLEMENTS, W., FOERSTER, J., AND LVOVSKY, A. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2020), vol. 34, pp. 3243–3250.

[8] BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 1 ed. Springer, 2007.

[9] BROCKMAN, G., CHEUNG, V., PETTERSSON, L., SCHNEIDER, J., SCHULMAN, J., TANG, J., AND ZAREMBA, W. Openai gym, 2016.

[10] CAI, Q., HANG, W., MIRHOSEINI, A., TUCKER, G., WANG, J., AND WEI, W. Reinforcement learning driven heuristic optimization. *Workshop on Deep Reinforcement Learning for Knowledge Discovery (DRL4KDD) abs/1906.06639* (2019).

[11] CAPPART, Q., MOISAN, T., ROUSSEAU, L.-M., PRÉMONT-SCHWARZ, I., AND CIRE, A. A. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), vol. 35, pp. 3677–3687.

[12] CHOWDHURY, M. R., MAHMUD, M. R., AND RAHMAN, R. M. Implementation and performance analysis of various vm placement strategies in cloudsim. *Journal of Cloud Computing 4* (2015), 1–21.

[13] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), pp. 273–286.

[14] DAVIES, A. P., AND BISCHOFF, E. E. Weight distribution considerations in container loading. *European Journal of Operational Research 114*, 3 (May 1999), 509–527.

[15] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805v2* (2018).

[16] DÓSA, G., AND SGALL, J. First fit bin packing: A tight analysis. In *30th International symposium on theoretical aspects of computer science (STACS 2013)* (2013), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[17] DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., WEISSENBORN, D., ZHAI, X., UNTERTHINER, T., DEHGHANI, M., MINDERER, M., HEIGOLD, G., GELLY, S., USZKOREIT, J., AND HOULSBY, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations* (2021).

[18] DUAN, L., HU, H., QIAN, Y., GONG, Y., ZHANG, X., WEI, J., AND XU, Y. A multi-task selected learning approach for solving 3d flexible bin packing problem. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems* (Richland, SC, 2019), AAMAS '19, International Foundation for Autonomous Agents and Multiagent Systems, p. 1386–1394.

[19] ELF, M., GUTWENGER, C., JÜNGER, M., AND RINALDI, G. Branch-and-cut algorithms for combinatorial optimization and their implementation in abacus. In *Computational Combinatorial Optimization*. Springer, 2001, pp. 157–222.

[20] ETHEVE, M., ALÈS, Z., BISSUEL, C., JUAN, O., AND KEDAD-SIDHOUM, S. Reinforcement learning for variable selection in a branch and bound algorithm. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (2020), Springer, pp. 176–185.

[21] FACEBOOK AI RESEARCH. Pytorch: An open source machine learning framework. https://pytorch.org/, 2019. Accessed: April 23, 2023.

[22] GASSE, M., CHÉTELAT, D., FERRONI, N., CHARLIN, L., AND LODI, A. Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems 32* (2019).

[23] GUPTA, P., GASSE, M., KHALIL, E., MUDIGONDA, P., LODI, A., AND BENGIO, Y. Hybrid models for learning to branch. *Advances in neural information processing systems 33* (2020), 18087–18097.

[24] HA, C. T., NGUYEN, T. T., BUI, L. T., AND WANG, R. An online packing heuristic for the three-dimensional container loading problem in dynamic environments and the physical internet. In *Applications of Evolutionary Computation: 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part II 20* (2017), Springer, pp. 140–155.

[25] HADARY, O., MARSHALL, L., MENACHE, I., PAN, A., GREEFF, E. E., DION, D., DORMINEY, S., JOSHI, S., CHEN, Y., RUSSINOVICH, M., ET AL. Protean: Vm allocation service at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (2020), pp. 845–861.

[26] HAJ-ALI, A., HUANG, Q. J., XIANG, J., MOSES, W., ASANOVIC, K., WAWRZYNEK, J., AND STOICA, I. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems 2* (2020), 70–81.

[27] HE, T., TAN, X., XIA, Y., HE, D., QIN, T., CHEN, Z., AND LIU, T.-Y. Layer-wise coordination between encoder and decoder for neural machine translation. In *Advances in Neural Information Processing Systems* (2018), S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc.

[28] HENDERSON, P., ISLAM, R., BACHMAN, P., PINEAU, J., PRECUP, D., AND MEGER, D. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence* (2018), vol. 32.

[29] HENDRYCKS, D., AND GIMPEL, K. Gaussian Error Linear Units (GELUs).

[30] HINTON, G., VINYALS, O., AND DEAN, J. Distilling the knowledge in a neural network, 2015. cite arxiv:1503.02531Comment: NIPS 2014 Deep Learning Workshop.

[31] HU, H., ZHANG, X., YAN, X., WANG, L., AND XU, Y. Solving a new 3d bin packing problem with deep reinforcement learning method, 2017.

[32] HUANG, S., DOSSA, R. F. J., YE, C., BRAGA, J., CHAKRABORTY, D., MEHTA, K., AND ARAÚJO, J. G. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research 23*, 274 (2022), 1–18.

[33] KAKADE, S. M. A natural policy gradient. *Advances in neural information processing systems 14* (2001).

[34] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[35] LEVINE, S., KUMAR, A., TUCKER, G., AND FU, J. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *ArXiv abs/2005.01643* (2020).

[36] LI, D., REN, C., GU, Z., WANG, Y., AND LAU, F. Solving packing problems by conditional query learning, 2020.

[37] LI, X., YUAN, M., CHEN, D., YAO, J., AND ZENG, J. A data-driven three-layer algorithm for split delivery vehicle routing problem with 3d container loading constraint. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining* (New York, NY, USA, 2018), KDD '18, Association for Computing Machinery, p. 528–536.

[38] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[39] MAI, V., MANI, K., AND PAULL, L. Sample efficient deep reinforcement learning via uncertainty estimation. In *International Conference on Learning Representations* (2022).

[40] MAO, H., SCHWARZKOPF, M., VENKATAKRISHNAN, S. B., MENG, Z., AND ALIZADEH, M. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*. 2019, pp. 270–288.

[41] MÜLLER, R., KORNBLITH, S., AND HINTON, G. E. When does label smoothing help? In *Advances in Neural Information Processing Systems* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc.

[42] NARAYANAN, D., KAZHAMIAKA, F., ABUZAID, F., KRAFT, P., AGRAWAL, A., KANDULA, S., BOYD, S., AND ZAHARIA, M. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 521–537.

[43] PADBERG, M., AND RINALDI, G. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review 33*, 1 (1991), 60–100.

[44] PANIGRAHY, R., TALWAR, K., UYEDA, L., AND WIEDER, U. Heuristics for vector bin packing. *research. microsoft. com* (2011).

[45] PARDO, F., TAVAKOLI, A., LEVDIK, V., AND KORMUSHEV, P. Time limits in reinforcement learning, 2018.

[46] PARREÑO, F., ALVAREZ-VALDÉS, R., TAMARIT, J. M., AND OLIVEIRA, J. F. A maximal-space algorithm for the container loading problem. *INFORMS Journal on Computing 20*, 3 (2008), 412–422.

[47] PETERSEN, B. K., LARMA, M. L., MUNDHENK, T. N., SANTIAGO, C. P., KIM, S. K., AND KIM, J. T. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations* (2021).

[48] PUTERMAN, M. L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[49] QI, M., WANG, M., AND SHEN, Z.-J. Smart feasibility pump: Reinforcement learning for (mixed) integer programming. *arXiv preprint arXiv:2102.09663* (2021).

[50] RADFORD, A., WU, J., CHILD, R., LUAN, D., AMODEI, D., AND SUTSKEVER, I. Language models are unsupervised multitask learners.

[51] RAFFEL, C., SHAZEER, N., ROBERTS, A., LEE, K., NARANG, S., MATENA, M., ZHOU, Y., LI, W., AND LIU, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research 21*, 140 (2020), 1–67.

[52] RAJESWARAN, A., GHOTRA, S., RAVINDRAN, B., AND LEVINE, S. EPOpt: Learning robust neural network policies using model ensembles. In *International Conference on Learning Representations* (2017).

[53] RENGARAJAN, D., VAIDYA, G., SARVESH, A., KALATHIL, D., AND SHAKKOTTAI, S. Reinforcement learning with sparse rewards using guidance from offline demonstration. *arXiv preprint arXiv:2202.04628* (2022).

[54] SCAVUZZO, L., CHEN, F. Y., CHÉTELAT, D., GASSE, M., LODI, A., YORKE-SMITH, N., AND AARDAL, K. Learning to branch with tree mdps. *arXiv preprint arXiv:2205.11107* (2022).

[55] SCHULMAN, J., LEVINE, S., ABBEEL, P., JORDAN, M., AND MORITZ, P. Trust region policy optimization. In *ICML* (2015), PMLR.

[56] SCHULMAN, J., MORITZ, P., LEVINE, S., JORDAN, M., AND ABBEEL, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

[57] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[58] SONG, J., YUE, Y., DILKINA, B., ET AL. A general large neighborhood search framework for solving integer linear programs. *Advances in Neural Information Processing Systems 33* (2020), 20012–20023.

[59] SUN, H., CHEN, W., LI, H., AND SONG, L. Improving learning to branch via reinforcement learning.

[60] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 2818–2826.

[61] TANG, Y., AGRAWAL, S., AND FAENZA, Y. Reinforcement learning for integer programming: Learning to cut. In *International conference on machine learning* (2020), PMLR, pp. 9367–9376.

[62] THALHEIM, J., OKELMANN, P., UNNIBHAVI, H., GOUICEM, R., AND BHATOTIA, P. Vmsh: hypervisor-agnostic guest overlays for vms. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 678–696.

[63] VAMVOUDAKIS, K. G., AND LEWIS, F. L. Online actor–critic algorithm to solve the continuous-time infinite horizon optimal control problem. *Automatica 46*, 5 (2010), 878–888.

[64] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).

[65] WIERING, M. A., AND VAN OTTERLO, M. Reinforcement learning. *Adaptation, learning, and optimization 12*, 3 (2012), 729.

[66] XIA, Y., TSUGAWA, M., FORTES, J. A., AND CHEN, S. Large-scale vm placement with disk anti-colocation constraints using hierarchical decomposition and mixed integer programming. *IEEE Transactions on Parallel and Distributed Systems 28*, 5 (2016), 1361–1374.

[67] ZHANG, J., ZI, B., AND GE, X. Attend2pack: Bin packing through deep reinforcement learning with attention. *ArXiv abs/2107.04333* (2021).

[68] ZHAO, H., SHE, Q., ZHU, C., YANG, Y., AND XU, K. Online 3d bin packing with constrained deep reinforcement learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021* (2021), AAAI Press, pp. 741–749.

[69] ZHU, H., GUPTA, V., AHUJA, S. S., TIAN, Y., ZHANG, Y., AND JIN, X. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 258–271.

[70] ZHU, Q., LI, X., ZHANG, Z., LUO, Z., TONG, X., YUAN, M., AND ZENG, J. Learning to pack: A data-driven tree search algorithm for large-scale 3d bin packing problem. In *Proceedings of the 30th ACM International Conference on Information Knowledge Management* (New York, NY, USA, 2021), CIKM '21, Association for Computing Machinery, p. 4393–4402.

[71] ZOPH, B., AND LE, Q. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations* (2017).

A Appendix

A.1 Background on (Deep) Reinforcement Learning

In this section, we give a brief overview on (deep) reinforcement learning while referring the readers to [5, 65] for a more detailed introduction.

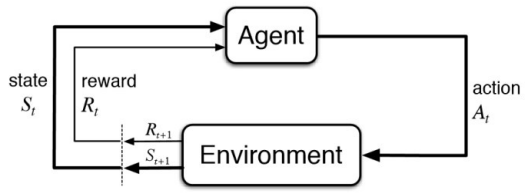


Figure 17: Illustration Diagram of Reinforcement Learning

Reinforcement Learning A standard Reinforcement Learning (RL) problem is typically characterized by a Markov Decision Process (MDP), where an agent continuously interacts with its environment, as depicted in Figure 17. At each step (or period), the agent observes a state s_k from the environment and responds with an action a_k in accordance with the current policy π, i.e., $a_k \sim \pi(\cdot|s_k)$. The agent subsequently receives a reward $r_k = r(s_k, a_k)$ as feedback, based on the state and action. The transition to the next state depends solely on the current state and action, i.e., $s_{k+1} \sim \mathbb{P}(\cdot|s_k, a_k)$, where $\mathbb{P}(\cdot|\cdot,\cdot)$ represents the transition probability. In a tabular RL setting, both the state space S and the action space A are assumed to be finite and discrete.

Commencing with the initial state s_0, the primary goal of the agent is to learn a policy that optimizes the so-called value function. This function is defined as the expected cumulative rewards received over K periods, i.e.,

$$J^{\pi}(s_0) := \mathbb{E}\left[\sum_{k=0}^{K} r(s_k, a_k)\middle|\pi, s_0\right]. \tag{10}$$

Here, the expectation is taken over all possible trajectories under policy π. Additionally, for any initial state-action pair (s_0, a_0), the corresponding state-action value function (Q-function) is defined as

$$Q^{\pi}(s_0, a_0) := \mathbb{E}\left[\sum_{k=0}^{K} r(s_k, a_k)\middle|\pi, (s_0, a_0)\right]. \tag{11}$$

Deep Reinforcement Learning Deep RL, a subset of RL, integrates RL and deep learning. In tabular RL, the policy is directly optimized as a table of dimensions $|S| \times |A|$. However, the exponential scaling of $|S|$ and $|A|$ in high-dimensional state and action spaces makes traditional RL algorithms impractical. Deep RL addresses this issue by employing (deep) neural networks to represent the policy function (or other learned functions) and developing specialized algorithms for scalability.

A.2 More Details on Risk-seeking Evaluation

A.2.1 Risk-seeking Evaluation

For some given (and fixed) initial state, let $\mathrm{FR}(\tau)$ denote the fragment rate solved by VMR^{2}L, where τ corresponds to the realized trajectory under the trained policy. Due to the randomized nature of the policy, $\mathrm{FR}(\tau)$ can be viewed as a random variable with finite mean and variance. Therefore, we can derive the following lemma about the minimum fragment rate obtained versus the number of sampled trajectories. We note that, since $\Phi(\cdot)$ is monotone increasing from 0 to 1, Equation 12 implies that the density of $\min_{1\le i\le k}\mathrm{FR}(\tau_i)$ is increasingly right-skewed as the number of sampled trajectories gets larger. This analysis shares insights on how many trajectories to sample at deployment time from a theoretical perspective.

Lemma A.1 *Suppose $p_0 = \mathbb{P}(\mathrm{FR}(\tau) < z) > 0$ for some $z \in (0,1)$. Let $\{\tau_i\}_{i=1}^k$ be k independently generated trajectories under the same policy. Then, with probability $1-(1-p_0)^k$, it holds that $\min_{1\le i\le k}\mathrm{FR}(\tau_i) \le z$. In particular, if $\mathrm{FR}(\tau)$ is normally distributed with mean μ and variance σ^2 as shown in Figure 25 in the Appendix, $\mathrm{FR}(\tau)$ has a density function $f(z)$ that satisfies the following expression:*

$$f(z) = \frac{k}{\sigma}\cdot\phi\left(\frac{z-\mu}{\sigma}\right)\cdot\left[1-\Phi\left(\frac{z-\mu}{\sigma}\right)\right]^{k-1}, \tag{12}$$

where $\phi(\cdot)$ and $\Phi(\cdot)$ are the density and cumulative distribution functions of standard normal distribution, respectively.

Proof of Lemma A.1. Due to independence, we have that

$$\begin{aligned}
&\mathbb{P}(\min_{1\le i\le k}\mathrm{FR}(\tau_i)\ge z)\\
&=\mathbb{P}(\mathrm{FR}(\tau_1)\ge z,\mathrm{FR}(\tau_2)\ge z,\ldots,\mathrm{FR}(\tau_k)\ge z)\\
&=\mathbb{P}(\mathrm{FR}(\tau_1)\ge z)\cdot\mathbb{P}(\mathrm{FR}(\tau_2)\ge z)\cdots\mathbb{P}(\mathrm{FR}(\tau_k)\ge z)\\
&=(1-p_0)^k,
\end{aligned} \tag{13}$$

which implies that $\mathbb{P}(\min_{1\le i\le k}\mathrm{FR}(\tau_i)\le z) = 1-(1-p_0)^k$.

When $\mathrm{FR}(\tau)$ is normally distribued with mean μ and variance σ^2, it further holds that

$$\begin{aligned}
&\mathbb{P}(\min_{1\le i\le k}\mathrm{FR}(\tau_i)\le z)\\
&=1-\prod_{i=1}^{k}\mathbb{P}(\mathrm{FR}(\tau_i)\ge z)\\
&=1-\prod_{i=1}^{k}\mathbb{P}\left(\frac{\mathrm{FR}(\tau_i)-\mu}{\sigma}\ge\frac{z-\mu}{\sigma}\right)\\
&=1-\left[1-\Phi\left(\frac{z-\mu}{\sigma}\right)\right]^k,
\end{aligned} \tag{14}$$

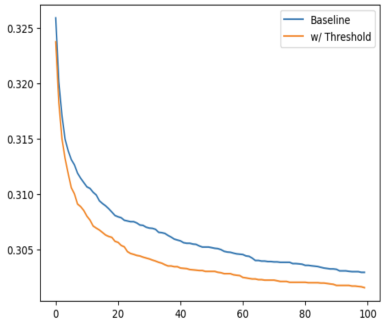


Figure 18: We test the effect of masking out low probability actions with and without thresholding. Here, we vary the number of sampled trajectories from 1 to 100. We see that the advantage of action thresholding increases quickly as we sample more trajectories. Intuitively, the more trajectories we sample, the smaller the improvement should become, as both versions would converge to a near-optimal solution. However, we see that the gap is still significant even when we sample 100 trajectories.

where we use independence again in the last line. Then, it follows that

$$\begin{aligned} f(z) &= \frac{\mathrm{d}\,\mathbb{P}(\min_{1\le i\le k} \mathrm{FR}(\tau_i) \le z)}{\mathrm{d}z} \\ &= \frac{k}{\sigma}\cdot\phi\left(\frac{z-\mu}{\sigma}\right)\cdot\left[1-\Phi\left(\frac{z-\mu}{\sigma}\right)\right]^{k-1}. \end{aligned} \tag{15}$$

□

A.2.2 Action Thresholding

The following lemma implies that, under the optimal policy, taking any action with non-zero probability yields the same expected total reward. However, due to approximation errors, the learned policy function would not be exactly equal to the optimal policy, even when the learning process does not get stuck on sub-optimal points. We expect those actions with significantly low probability in the learned policy π are caused by approximation errors and are indeed sub-optimal.

Lemma A.2 *For any state s, if there exists actions a and a' such that $\pi^\star(a|s) > 0$ and $\pi^\star(a'|s) > 0$, it follows that $Q^{\pi^\star}(s,a) = Q^{\pi^\star}(s,a')$, i.e., taking either action a or a' in the first period yields the same expected total reward under the optimal policy $\pi^\star$.*

Proof of Lemma A.2. By definition, we have the following relation between the value function and the Q-function under the optimal policy $\pi^\star$:

$$J^{\pi^\star}(s) = \sum_{a\in A} \pi^\star(a|s)\cdot Q^{\pi^\star}(s,a). \tag{16}$$

On the other hand, the Bellman equation [48] implies that

$$J^{\pi^\star}(s) = \max_{a\in A} Q^{\pi^\star}(s,a). \tag{17}$$

Let $A(s) = \{a \in A | \pi^\star(a|s) > 0\}$. The two equations above together imply that $A(s) \subseteq \arg\max_{a\in A} Q^{\pi^\star}(s,a)$ and $Q^{\pi^\star}(s,a) = Q^{\pi^\star}(s,a'), \forall a,a' \in A(s)$. □

A.3 Risk-seeking Training

Traditional policy gradient methods, such as Proximal Policy Optimization (PPO) [57], optimize for high expected rewards during training by optimizing the value function in Equation 10 using the policy gradient defined as:

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim p(\tau|\theta)}\left[R(\tau)\nabla_\theta \log p(\tau|\theta)\right] \\ &\approx \frac{1}{N}\sum_{i=1}^{N} R(\tau_i)\nabla_\theta \log p(\tau_i|\theta), \end{aligned} \tag{18}$$

where τ denotes the trajectory and θ represents the policy parameter.

However, in VM rescheduling, we are primarily concerned with best-case performance. This creates a discrepancy between our training objective and evaluation criteria. To address this gap, we propose training our RL agent with a risk-seeking objective [47]. Under the same policy, we can reset to the same initial mapping and sample multiple trajectories with identical initial states. Specifically, we can randomly sample N initial mappings, run K trials on each, and retain only the best-performing trial for each initial mapping during training. Then, we only utilize these best-performing trajectories during the computation of the policy gradient:

$$\begin{aligned} \nabla_\theta J_{\text{Ours}}(\theta;\varepsilon) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}&\left[R(\tau_{i,k}) - b\right] \\ &\cdot\left[\mathbf{1}_{R(\tau_{i,k})=\max R(\tau_{i,\cdot})}\right]\nabla_\theta \log p(\tau_{i,k}\mid\theta), \end{aligned} \tag{19}$$

where b is the critic score, and the indicator function $\mathbf{1}_{R(\tau i,k)=\max R(\tau_{i,\cdot})}$ ensures that we only train on the best trajectory out of the K trajectories for each initial mapping.

This risk-seeking approach has also been adopted in existing works, such as neural architecture search [71] and symbolic regression [47]. The method EPOpt-ε by [52] integrates the concept of Conditional Value at Risk (CVaR) into policy gradient training by training only on a portion of the N samples that yield the lowest reward. Conversely, the algorithm DSR by [47] optimizes for best-case performance and proposes leveraging only the top-performing samples.

A.4 Training Algorithm

The algorithm VMR^{2}L is developed based on the Proximal Policy Optimization (PPO) algorithm [57]. Recognized for

Algorithm 1: Learning a VM rescheduling policy using a two-stage actor-critic algorithm.

Input: All VMs and PMs' State
Output: A VM rescheduling agent
1 Initialize the VM actor parameters θ_{vm} with pre-trained actor $\hat{\theta}_{vm}$ from Algorithm 2 ;
2 Initialize PM actor parameters θ_{pm} with random weights ;
3 Initialize critic parameters θ_v with random weights ;
4 **for** *episode =0,1,...,M* **do**
5 Obtain the initial state S_t and A_t randomly;
6 buffer.clear() ;
7 **for** *rescheduling time step t = 0,1,...,T* **do**
8 $logp_{vm} \leftarrow \pi(a|S_t, \theta_{vm})$;
9 $a_{vm} \leftarrow$ p$_{vm}$.sample() ;
10 $logp_{pm} \leftarrow \pi(a|S_t, a_{t,vm}, \theta_{pm})$;
11 $a_{pm} \leftarrow$ p$_{pm}$.sample() ;
12 $v \leftarrow$ V(S, θ_v) ;
13 buffer.append($logp_{vm}, logp_{pm}, a_{vm}, a_{pm}$) ;
14 **if** $EpisodeEnd(S)$ **then**
15 $S_{t+1} = ENV.Reset(S)$;
16 // Compute gradients wrt. *vm* actor, *pm* actor, and critic gradient loss ;
17 $d\theta_{vm}, d\theta_{pm}, d\theta_v \leftarrow ComputeLoss(buffer)$;
18 Perform update of θ_{vm} using $d\theta_{vm}$ and θ_{pm} using $d\theta_{pm}$ and θ_v using $d\theta_v$;

its stability and robustness to various hyperparameters and network architectures [57], PPO has exhibited superior performance compared to Natural Policy Gradients (NPG) [33] and Trust Region Policy Optimization (TRPO) [55], and exhibited less bias compared to Q-learning [63]. Despite PPO's high sample complexity, it becomes less of a concern due to our cost-effective simulator as detailed in Section 4. In VMR^{2}L, we employ two actors — a VM actor $\pi_{vm}(a_{vm}|s;\theta_{vm})$ and a PM actor $\pi_{pm}(a_{pm}|s, VM;\theta_{pm})$. Given the current state encoding s, the VM actor samples the next-step *vm* action, denoted as a_{vm}. Based on both s and a_{vm}, the PM actor samples a destination PM from the *pm* distribution, denoted as a_{pm}. Then, the critic $V(s;\theta_v)$ outputs an evaluation for the current state embedding s.

Below, we provide a brief overview for the training algorithm, with the complete pseudocode deferred to Algorithm 1 in the appendix. In Lines 1-3, the algorithm first initializes the parameters of the VM actor with the weights, $\hat{\theta}_{vm}$, obtained from the pre-trained VM actor. The PM actor and the critic are initialized with random weights. To simplify implementation, we incorporate the critic network into the VM actor by appending a special VM with all '-1' features and using its output as the critic score.

Each episode is made up of *MNL* steps, with each step

Algorithm 2: Pretraining with Distillation.

Input: VM actor $\pi(\text{score}|S_t, \hat{\theta}_{vm})$, heuristic VM model $\pi_{\text{vm\_teacher}}(\text{score}|S_t)$, greedy PM model $\pi_{\text{pm}}(a|S_t)$
Output: A pre-trained VM actor, $\pi(a|S_t, \hat{\theta}_{vm})$
1 Initialize VM actor $\hat{\theta}_{vm}$ with random weights;
2 Initialize dataset $\mathcal{D} \leftarrow \emptyset$;
3 **for** *collect_times i = 0, 1, ⋯, M* **do**
4 Sample a random initial state $S_{i,0}$;
5 **for** *reschedule_times t = 0,1,...,T* **do**
6 score$_{i,t} \leftarrow \pi_{\text{vm_teacher}}(S_{i,t})$
7 $a_{vm} \leftarrow$argmax$(score_{i,t})$
8 $a_{pm} \leftarrow \pi_{\text{pm}}(S_{i,t})$
9 Execute actions $S_{i,t+1} \leftarrow$Env(S$_{i,t}$, (a_{vm}, a_{pm}))
10 Append to dataset $\mathcal{D} \leftarrow$D $\bigcup(S_{i,t}$, score$_{i,t})$
11 **for** *update_iterations* $j = 0, 1, \cdots, N$ **do**
12 Sample $(S_j, score_j)$ from $\mathcal{D}$
13 score$_{pred} \leftarrow \pi(a|S_j, \hat{\theta}_{vm})$
14 $\hat{\theta}_{vm} \leftarrow \hat{\theta}_{vm} - \alpha \bigtriangledown_\theta \ell_{MSE}(score_{pred}, score_j)$;

representing a rescheduling action. During every rescheduling time step, VMR^{2}L commences *vm* and *pm* selection with the original *vm-pm* mapping. It then generates the new *vm-pm* mapping by iteratively executing an action, as computed by the *vm* and *pm* actors, on the current *vm-pm* mapping until the rescheduling time step concludes (Line 6-16). The episode is terminated if the rescheduling time step length exceeds a pre-determined threshold.

Upon the completion of each episode, we compute the gradient for both actors and critics using their loss functions (Line 17). The gradient loss of the *vm* and *pm* actors is defined as the mean error between the advantage estimate and the logit of the corresponding sampled *vm* and *pm* actions ($logp_{vm}$ and $logp_{pm}$) across the episode. The Generalized Advantage Estimate for step i (GAE$_i$) is calculated as per the GAE-Lambda advantage [56] by

$$GAE_i = r_i + \gamma \cdot v_{i+1} - v_i + \gamma \cdot \lambda \cdot GAE_{i+1}, \tag{20}$$

where r_i, v_i represent the reward and the critic's output at step i respectively. γ is the discount factor and λ is a smoothing parameter for variance reduction. The critic gradient loss is defined as the mean-square error between the reward to go and the critic's output, v, across the epoch. The reward-to-go is calculated by applying the discount factor to the intermediate rewards. Finally, in Line 18, we update the parameters of the actor and critic networks with the calculated gradients.

A.5 Sparse Attention Details

In this section, we delve into the detailed formulation of *sparse attention*, initially introduced in Section 3.3.2.

Conceptual Overview. Fundamentally, attention can be understood as a trainable dictionary involving queries, keys, and values. Given an embedding vector (a VM or a PM) that requires an update, and a set of reference vectors (selected VMs and/or PMs), we project the vector-to-be-updated into a query vector. Concurrently, we project all reference vectors into key and value vectors. We then compute the similarity score between the query vector and each key vector. Based on these scores, we update the target embedding vector as the weighted sum of the corresponding values.

The term "VM self-attention" refers to the process of updating each VM's embedding vector using all VM's embedding vectors (including its own) as a reference. "PM self-attention" operates similarly for PMs. "VM-PM attention" involves updating each VM's embedding vector using all PM's embedding vectors as references. Lastly, the proposed tree-level *sparse attention* involves updating each VM or PM using only other VMs or PMs within the same tree—that is, those affiliated with the same PM.

Attention Formulation. Formally, let $\mathcal{V} = \{v_i\}$ and $\mathcal{P} = \{p_i\}$ denote the set of feature vectors for each VM and each PM, respectively. Consider an embedding vector $x_i \in \mathbb{R}^{d_{\text{model}}}$ that we aim to update, which could either be $v \in \mathcal{V}$ or $p \in \mathcal{P}$. Let $(y_1, \cdots, y_n)$ be a set of reference vectors that could be a combination of $v \in \mathcal{V}$ and $p \in \mathcal{P}$, including x_i itself. Instead of directly operating in the feature space $\mathbb{R}^{d_m}$, we project these vectors into an embedding space $\mathbb{R}^{d_k}$.

An attention function updates the target vector x_i by first projecting it into a query $Q_i = x_i W^Q$ using a linear transformation, where $W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ is a learnable weight matrix. Similarly, each reference vector y_j is projected into a key $K_j = y_j W^K$ and a value $V_j = y_j W^V$, where $W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$[8]. Note that we can perform the attention function on a set of queries in parallel by consolidating Q_i's into a larger matrix Q[9].

The compatibility function, measuring the similarity between Q_i and K_j, is typically implemented as the dot product of these two vectors. However, when d_k is large, the magnitude of dot products tends to increase, leading to a high similarity for only a few keys and a marginal similarity for the rest of the references. This has been known to cause extremely small gradients [64]. To address this, we scale the dot product by $\frac{1}{\sqrt{d_k}}$,

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{21}$$

For the proposed *sparse attention*, we introduce an additional mask M. Here, $M_{i,j} = -\infty$ if x_i and y_j are not part of the same PM tree and zero otherwise.

$$\text{Sparse-Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V. \tag{22}$$

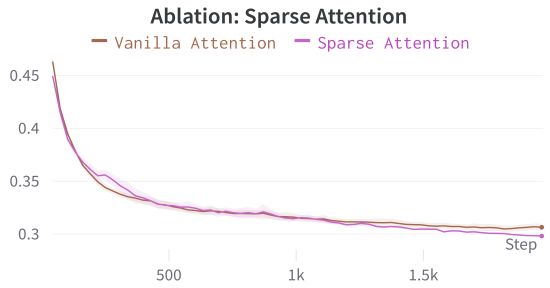


Figure 19: FR Performance of VMR^{2}L with Sparse versus Vanilla Attention.

Attention Block. In actual applications, we often utilize multiple sets of W^Q, W^K, and W^V. Each set, referred to as a single attention head, enables the model to extract pertinent information from a distinct representation space. We combine the resulting matrices from each attention head in multi-head attention by concatenation.

$$\text{Multi-Head}(Q,K,V) = \text{Concat}\left(\text{head}_1, \cdots, \text{head}_h\right)W^O, \tag{23}$$

where head_i represents the output of each attention head from Equation 22 using W_i^Q, W_i^K, and W_i^V. $W^O \in \mathbb{R}^{h \cdot d_k \times d\text{model}}$ is an additional learnable weight parameter.

Within each attention block, we consecutively execute the process for tree-level *sparse attention*, VM/PM self-attention, and VM-PM attention. Following these three attention submodules, we further process each updated embedding vector with a fully connected feed-forward network, which operates independently on each embedding vector.

$$\text{FFN}(x_i) = \text{GELU}\left(xW_1 + b_1\right)W_2 + b_2, \tag{24}$$

where GELU refers to the Gaussian Error Linear Units activation function [29]. The output of this process is then supplied to the next attention block, and the process is iteratively repeated. For more in-depth details regarding the attention module, we encourage readers to consult the paper by Vaswani et al. [64].

Overall Architecture. The architecture incorporates the remaining *MNL* steps as supplementary input for both the VM and PM actors, processed concurrently with the PMs. This inclusion is pivotal for reinforcement learning in scenarios with fixed-length episodes as demonstrated in prior studies [45]. As for the critic, a node containing all -1's is incorporated and

[8] W^V can have different dimensions than W^K. We keep them the same here for simplicity.

[9] As we use the same W^Q, W^K, Q^V for all vectors, the total number of weight parameters remains independent of the number of VMs or PMs in the problem, making this approach suitable for scaling to large data centers.

processed alongside the VMs. The output embedding from the terminal block undergoes a linear projection to yield the critic score. It's important to note that the PM embeddings undergo iterative updates block-by-block in tandem with the VM embeddings. This approach fosters better coordination between PMs and VMs, enabling a gradual transition from low to high-level updates [27].

In Figure 19, we present an ablation study performed on a medium dataset with $MNL = 50$ to evaluate the impact of the *sparse attention* module in terms of FR. As a consequence of the supplementary parameters involved, *sparse attention* exhibits a longer convergence time but successfully achieves a lower FR. This observation validates that *sparse attention* empowers VMR^{2}L to extract tree-level information, which is absent in vanilla attention. A visual illustration of this concept is provided in Section A.8.

A.6 Fragment Rate on Individual Datasets.

Figure 20 illustrates the FR performance of GA, VMR^{2}L and MIP tested on 200 datasets. The x-axis shows the initial FR of each test dataset, and the y-axis shows the final FR achieved by three methods. Figure 20(a) shows the FR performance at MNL = 10. Given the initial average FR of 0.555, all three methods can reduce the FR to around 0.485 after rescheduling, since simply applying the greedy approach is enough to guarantee good performance at the early stages. However, in Figure 20(b) and 20(c), we see that GA scatters to the upper part and the other two methods (VMR^{2}L and MIP) scatter to the lower part. The average FR are 0.387, 0.2953, 0.2874 at MNL = 50 for GA, VMR^{2}L and MIP respectively. This means that the performance gap between GA and the other two (VMR^{2}L and MIP) methods gradually increases for larger MNLs, since GA only optimizes for one-step ahead and thus quickly depletes the available rescheduling actions. In comparison, the performance gap between VMR^{2}L and MIP stays at a low level of 2.67%.

A.7 Simulator Cross-Validation

VMR^{2}L's training happens offline using a faithful simulator that has access to profiling information (e.g., VM-PM mapping) from a real cloud data center. To faithfully simulate how VMR^{2}L 's decisions interact with a real cloud data center, we validate the simulator via the following steps:

1. We generate ten steps of VM rescheduling actions with an in-house MIP solver.
2. We obtain the resulting VM-PM mapping from the MIP solver.
3. We take the same actions with the VM Rescheduling Simulator and calculate the resulting FR.

Figure 21 shows that the VM Rescheduling Simulator can exactly follow MIP's solution and simulate 100% correct FR changes.

A.8 A Case Study

We randomly select one dataset from 200 test datasets and analyze how GA, MIP, and VMR^{2}L reduce FR, when MNL is set to 50. Figure 22 shows the FRs at each step when VMR^{2}L migrates one VM. GA cannot further decrease FR after migrating 23 VMs. MIP and VMR^{2}L can decrease FR by migrating more VMs. GA only considers the best VM that can reduce the FR at each step. However, MIP and VMR^{2}L consider optimizing FR from a global point of view.

The red square in Figure 22 illustrates how VMR^{2}L reschedules VMs to reduce FR at steps 38, 39, 40 and 41. We study the behavior of four PMs (PM118, PM83, PM139 and PM86) that participate in the rescheduling at these four steps. For each PM, we draw two bars for the two NUMAs on this PM and all the VMs with different colors on each NUMA. The number on each bar is the fragment ($CPU_{remaining}\%16$) on each NUMA. The height is the total CPU (44) on each NUMA. The bigger number means a high FR on each NUMA. In step 38, the sum of all FR in these four PMs is 32, VMR^{2}L decides to migrate VM from its source PM118 to PM83 and go to step 39, we can see the FR is 48 and bigger than step 38. Then, VMR^{2}L migrates VM870 from PM118 to PM139 and goes to step 40, we find that the FR is reduced to 32. In step 40, the VMR^{2}L migrates VM1336 from PM83 to PM86 and goes to step 41, we can see that VMR^{2}L achieves the best FR 16 among in four steps. VMR^{2}L sacrifices immediate FR performance for long-term FR performance by leveraging the neural network through accumulated rewards.

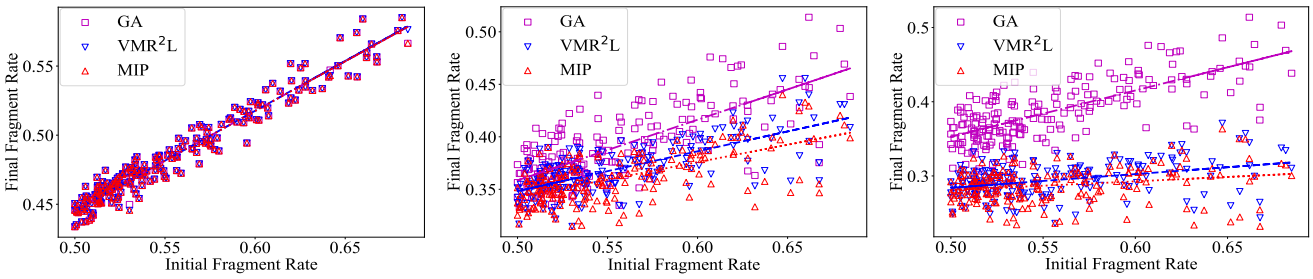
A.9 Experiment Details

A.9.1 Datasets

We have two seed initial mappings from an industry-scaled real cloud data center – one medium dataset with 2089 VMs and 280 PMs, and one large dataset with 4546 VMs and 1176 PMs. Note that the RL agent must be able to generalize to VM-PM mappings unseen during training and a dynamic number of VMs at deployment time. To better evaluate the agent's performance under various initial mappings, we generate 4400 initial mappings with different numbers of VMs for both the medium and the large datasets. Each mapping is generated by removing the existing VMs on each PM and randomly scheduling some of them to any PM that can fit them. We generate three versions of the middle dataset with low, middle, and high workloads (different remaining CPU resources). We leverage 4000 datasets for training, 200 datasets for both validation and test. Both the simulator and datasets are available to the research community.

A.9.2 Different Service Constraints

Multi-Resource. As the VM types we have in-house all have more CPUs than memory as shown in Table 1, we collect a



(a) MNL=10 (b) MNL=30 (c) MNL=50

Figure 20: Fragment rate performances on 200 datasets at MNL $= \{10, 30, 50\}$.

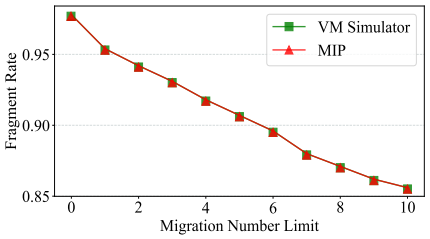


Figure 21: Cross Validation on the VMR Simulator.

new dataset with a wide variety of VM and PM types. We include two PM types — the first type has 88 CPUs and 256 GB of memory, and the second type has 128 CPUs and 364 GB of memory. The regular VM types always have a CPU-memory ratio of 1 : 2, except for 22xlarge which takes up an entire PM. For memory-intensive applications, a user might request for additional memory so that CPU-memory ratio becomes 1 : 4 or 1 : 8. We denote 1 : 4 as double VMs and 1 : 8 as quadruple VMs. We augment each of the VM types in Table 1 except for Quadruple 22xlarge as no PM can handle this VM type.

Service Anti-Affinity. To synthesize the service anti-affinity constraint of each VM on the medium dataset, we must make sure that the VM has no conflicts with any other VMs deployed on the same PM in the initial mappings. To achieve this, we first construct small clusters in each group of neighboring PMs by only including one VM from each PM. All VMs that belong to the same cluster conflict with each other and cannot be placed on the same PM. We then iteratively merge neighboring clusters until a desired level of anti-affinity is obtained. However, we cannot construct arbitrary constraint levels using this process as we must not merge two clusters if they include VMs that are affiliated with the same PM. Therefore, at affinity level = 8, for each VM, we directly sample 800 VMs from all VMs that do not belong to the same PM. The aforementioned process is used to construct all the remaining affinity levels.

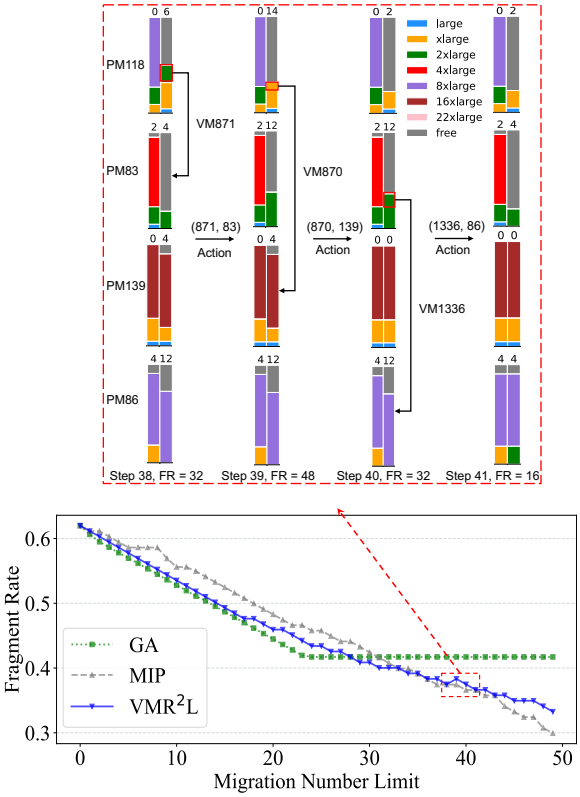
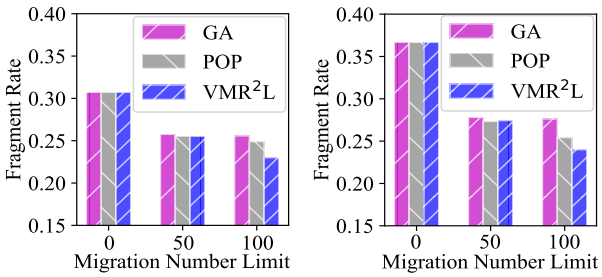


Figure 22: PM-VM Migration Details.

A.9.3 Different Workloads with Different MNLs

We evaluate VMR^{2}L with varying workloads under different MNLs. We set MNL=100 when evaluating the Low and Middle workload dataset since the FR difference is low until increasing MNL to 100. From Figure 23, we can see GA, POP, and VMR^{2}L decrease FR at MNL 50. However, GA fails to decrease FR at MNL 100. In comparison, VMR^{2}L achieves +7.42% and +4.8% on the low workload and +13.77% and +6.3% on the middle workload compared to GA and POP, respectively. These results demonstrate that VMR^{2}L is capable of handling varying workloads.



(a) FR on the low workload. (b) FR on the medium workload.

Figure 23: FR on Different Workloads.

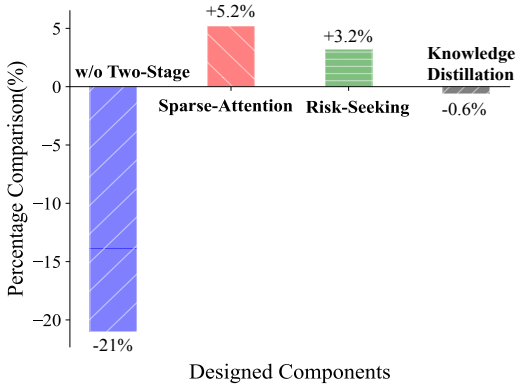


Figure 24: Performance Decomposition for VMR^{2}L.

A.9.4 Reward for Minimizing the MNL Given FR Goals

Equation 27 presents the reward function designed to minimize the MNL and achieve the desired FR goal. To incorporate the information from Equations 25 and 26, we include a penalty of -1 if the FR falls below the goal, as this indicates that additional VM migration is required to meet the goal. Additionally, we offer a bonus of +10 to incentivize the VMR^{2}L agent to approach the FR goal.

$$S_i = \sum_{j=0}^{1} \left(\tilde{U}_{i,j} \% X\right) \div c, \tag{25}$$

$$R_{fr} = (S_{\text{before, src}} - S_{\text{after, src}}) + (S_{\text{before, dest}} - S_{\text{after, dest}}), \tag{26}$$

$$R = \begin{cases} -1 + R_{fr}, & FR < FR\_Goal \\ 10 + R_{fr}, & FR = FR\_Goal \end{cases} \tag{27}$$

A.9.5 Hyperparameter Details

The VM and PM actors are based on the attention-based transformer model framework [64], which includes two encoders and two decoders. Each encoder and decoder has two heads in the multi-head attention models, with a feed-forward network model of 64 dimensions for each. The architecture of the VM candidates model is the same as the VM actor, and the weights from the VM candidates model can be used directly for the VM actor. The RL agent is trained using the ADAM optimizer [34], with a learning rate of 2.5e-4, and the experiments comprise a total of 200 million timesteps. The model of VMR^{2}L is lightweight and mapping the VM-PM state to a scheduling decision takes less than 150 ms, making it an efficient solution for VM rescheduling.

Table 4: Hyperparameters

RL Parameters	Value	General Parameters	Value
Clip_coefficient	0.1	Total_iterations	4e6
Discount_factor	0.99	Learning_rate	2.5e-4
PPO update_epochs	8	Attention_blocks	2
PPO minibatches	128	Attention_heads	2
GAE_Lambda	0.95	Transformer d_{ff}	32
Risk-Seeking K	3	Transformer d_{hidden}	64

A.9.6 FR and Running Time

The absolute evaluation numbers for Figure 9 in the paper are listed in Table A.9.4. For each column, the best results besides MIP are highlighted. We sample 8 trajectories for Decima and VMR^{2}L and report the best result.

A.10 Visualization Tool

We built a visualization tool that allows end-users to directly input a trained agent and receive the detailed migration actions at each step in the form of a gif file. This allows better intrepretation and trouble-shooting. We show an example in Figure A.10.

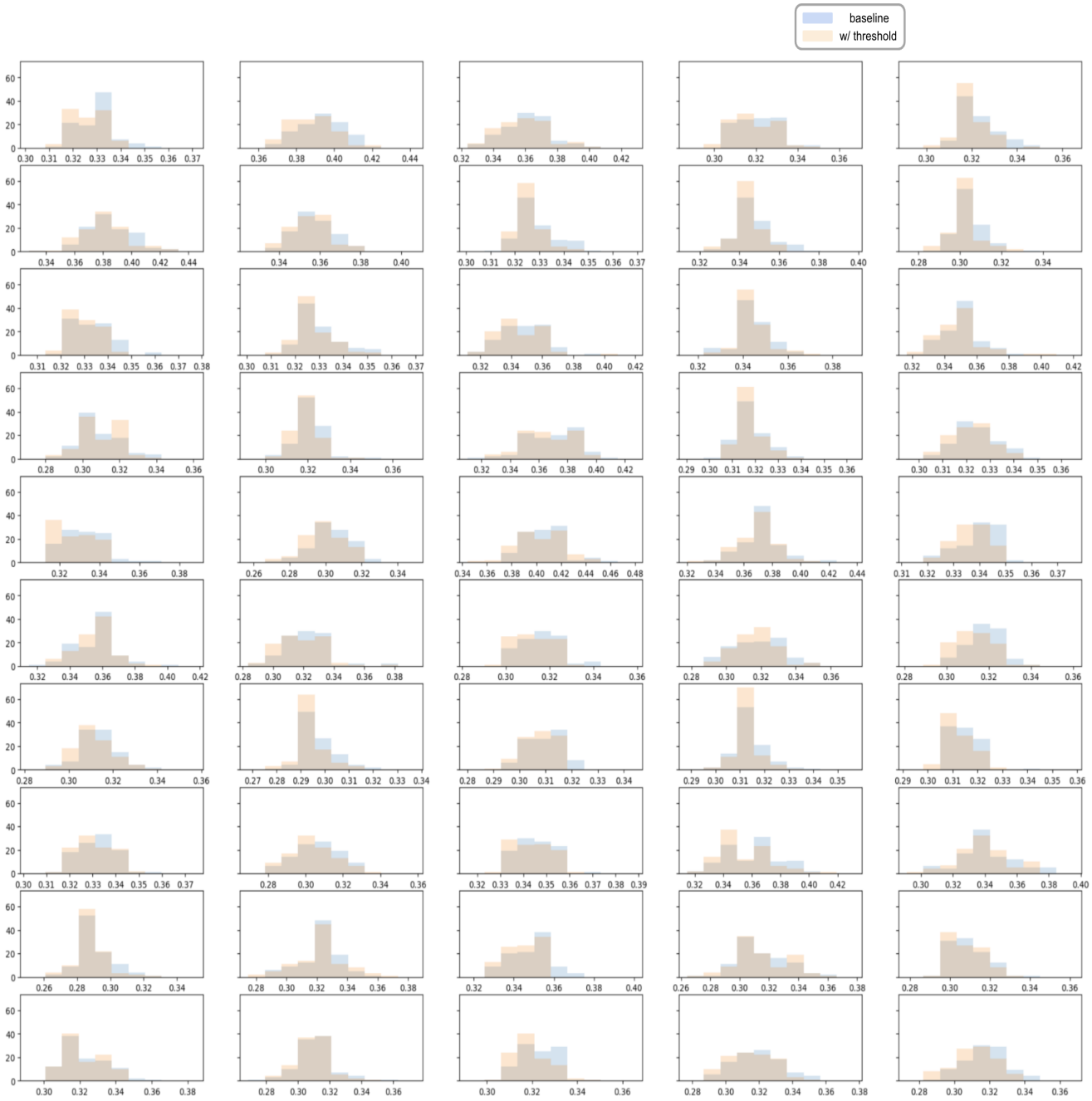


Figure 25: Distribution of final fragment rate when we sample 20 trajectories with and without thresholding on 50 different initial mappings. The threshold is set arbitrarily to mask out the low probability actions whose probabilities sum to approximately 0.5%. Here, the x-axis is the final FR. We see that masking out actions that are highly likely to be suboptimal makes it more likely to sample better performing trajectories. Indeed, the minimum fragment rate is 0.2953 compared to 0.2976 without thresholding.

Figure 26: We visualize two steps from VMR^{2}L. To better utilize the 8 fragments on NUMA0, the agent first remove a VM with 4 CPUs in Step 38 to help the destination NUMA to achieve zero fragments. Then in Step 39, it removes an additional VM with 4 CPUs and reschedule it to achieve zero fragments on both the source and the destination NUMAs. This is possible because the proposed *sparse attention* module allows each VM to exchange information with other VMs under the same tree.

Table 5: Fragment Rate & Solution Time(s)

Method	**MNL=10**		**MNL=20**		**MNL=30**		**MNL=40**		**MNL=50**	
	FR	**Time**	**FR**	**Time**	**FR**	**Time**	**FR**	**Time**	**FR**	**Time**
GA	**0.485**	0.18	0.420	0.24	0.390	0.23	0.387	0.22	0.387	0.24
α-VBPP	0.509	0.13	0.469	0.21	0.423	0.27	0.394	0.36	0.387	0.42
POP	0.555	0.57	0.468	1.90	0.468	1.93	0.399	2.07	0.345	1.94
MCTS	**0.485**	0.07	0.420	0.15	0.388	0.59	0.378	1.39	0.378	2.47
Decima	0.515	0.09	0.487	0.18	0.470	0.27	0.456	0.36	0.448	0.45
NeuPlan	**0.485**	18.06	**0.417**	35.70	0.396	35.31	0.372	35.20	0.358	34.80
VMR^2L	**0.485**	**0.03**	0.419	**0.06**	**0.369**	**0.09**	**0.330**	**0.12**	**0.295**	**0.15**
MIP	0.485	17.06	0.417	35.70	0.361	680	0.322	1380	0.2859	3033